Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Institut AutoMATH – MuPAD-**Forschungsgruppe**

Elementare Kryptographie

Kai Gehrs
acrowley@mupad.de

Paderborn, 10. Mai 2006

Kryptographie

In dem folgenden Teil des Skripts wollen wir uns mit kryptographischen Verfahren beschäftigen. In Zeiten des Internets, von Emails, von Online-Banking etc. ist es von zentraler Bedeutung, auch vertrauliche Daten über einen solchen offenen Kanal kommunizieren zu können. Dabei sollten diese Daten in einer verschlüsselten Form versendet werden, die es zwar dem berechtigten Empfänger der Nachricht erlaubt, die Verschlüsselung aufzuheben, nicht aber einem unberechtigten Dritten, der die Kommunikation ohne Wissen der Beteiligten abhört oder gar Nachrichten abfängt.
Es gibt viele gute Gründe, sich aus sicherheitstechnischen Erwägungen mit Kryptographie zu beschäftigen. Die Kryptographie ist aber insbesondere auch ein interdisziplinäres Feld in der Mathematik, das sich Resultaten aus Computeralgebra, Algebra, Zahlentheorie, Komplexitätstheorie u.v.m. bedient und so schön zeigt, wie sich verschiedene Zweige der Mathematik in einer neuen Theorie/Anwendung zusammenfinden.
Eines der bekannten kryptographischen Verfahren ist das RSA-Verfahren (benannt nach seinen Erfindern *R. L. Rivest*, *A. Shamir* und *L. M. Adleman*) und seine Kenntnis sollte fast schon zur mathematischen Allgemeinbildung gehören. Wir geben das Kommunikationsprotokoll als Kommunikation zweier Personen Alice und Bob an. Bob möchte Alice eine verschlüsselte Nachricht senden. Dafür trifft Alice zunächst die folgenden Vorbereitungen:

1) Alice wählt zufällig zwei verschiedene Primzahlen p und q.
2) Sie berechnet $N := p \cdot q$ und $\varphi(N) = (p-1) \cdot (q-1)$.
3) Dann wählt sie $e \in \{2, \dots, \varphi(N) - 2\}$ mit $\mathrm{ggT}(e, \varphi(N)) = 1$.
4) Alice bestimmt d mit $e \cdot d = 1 \bmod \varphi(N)$.
5) Dann ist (N, e) der öffentliche und (N, d) der geheime Schlüssel.
6) Die Werte p, q und $\varphi(N)$ löscht Alice.

Bob möchte Alice die Nachricht $x \in \{0, \dots, N-1\}$ schicken. Er geht nun wie folgt vor:

7) Bob berechnet $y := x^e \bmod N$ und schickt y an Alice.

8) Alice berechnet $x = y^d \bmod N$.

Wenn man das RSA-Verfahren nicht kennt, drängen sich an dieser Stelle natürlich viele Fragen auf:

- Wie wählt Alice zufällig Primzahlen?
- Welche Rolle spielen N und $\varphi(N)$?
- Wie berechnet man das Inverse des Elementes $e \bmod \varphi(N)$ bzw. wie rechnet man überhaupt "modulo" N oder $\varphi(N)$?
- Ist das Verfahren überhaupt effizient auf einem Computer implementierbar, wenn man z.B. p und q als 1024-Bit oder sogar 2048-Bit Primzahlen wählt?
- Wieso gilt $x = y^d \bmod N$?
- Warum ist das Entschlüsseln verschlüsselter Nachrichten für Personen, die den privaten Schlüssel (N, d) nicht kennen, schwierig?
- usw.

Die meisten der Fragen werden wir in den folgenden Abschnitten klären. Wir beginnen mit der Erarbeitung der notwendigen Grundlagen aus Algebra und Computeralgebra, machen dann "Abstecher" in die Zahlen- und Gruppentheorie und werden schließlich auf das RSA-Verfahren und einige weitere kryptographische Verfahren zurückkommen.

Kapitel 1

Grundlagen aus Algebra und Computeralgebra

In diesem Kapitel werden die grundlegenden theoretischen Hilfsmittel aus der Computeralgebra diskutiert, die nötig sind, um einige kryptographische Verfahren verstehen und deren Korrektheit einsehen zu können. Ferner werden wir Algorithmen zitieren und grob ihre Laufzeiten angeben, um begründen zu können, dass die vorgestellten Kryptosysteme für ihre berechtigten Benutzer effizient verwendet werden können. Laufzeiten werden wie üblich in der $\mathcal{O}$-Notation angegeben:

Definition 1.1. Seien $f, g : \mathbb{Z} \to \mathbb{R}$ Funktionen mit $f(n), g(n) \geq 0$ für alle $n \gg 0$. Dann gilt $g \in \mathcal{O}(f)$, falls $g(n) \leq c \cdot f(n)$ für alle $n \geq N$ mit $N \in \mathbb{N}$ und $c \in \mathbb{R}$.

Damit bedeutet $g \in \mathcal{O}(f)$ im wesentlichen nichts anderes, als dass g bis auf Multiplikation mit einer Konstanten $c \in \mathbb{R}$ "höchstens so schnell wächst" wie f.

1.1 Addition und Multiplikation

Wir werden uns in der Regel auf die Addition und Multiplikation von ganzen Zahlen beschränken. Jede ganze Zahl $x \in \mathbb{Z}$ können wir in *Binärdarstellung* in der Form $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$ mit $x_i \in \{0, 1\}$ für $i = 0, \ldots, n-1$ schreiben. Die Zahl n heißt dann die *Binärlänge* von x.

Lemma 1.2. *Addition zweier n-Bit Zahlen $x, y \in \mathbb{Z}$ ist mit $\mathcal{O}(n)$ Bit-Operationen durchführbar.*

Beweis. Wir schreiben $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$ sowie $y = \sum_{i=0}^{n-1} y_i \cdot 2^i$ mit $x_i, y_i \in \{0, 1\}$ für alle $i = 0, \ldots, n-1$. Addition von x und y wird komponentenweise auf den Bits durchgeführt. Daraus folgt die Behauptung. □

Lemma 1.3. *Multiplikation zweier n-Bit Zahlen $x, y \in \mathbb{Z}$ ist mit $\mathcal{O}(n^2)$ Bit-Operationen durchführbar.*

Beweis. Wir schreiben $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$ sowie $y = \sum_{i=0}^{n-1} y_i \cdot 2^i$ mit $x_i, y_i \in \{0, 1\}$ für alle $i = 0, \ldots, n-1$. Analog zur Multiplikation von Polynomen benötigt man $\mathcal{O}(n \cdot n) = \mathcal{O}(n^2)$ Bit-Operationen. □

Bemerkung 1.4. Es sind schnellere Methoden zur Multiplikation zweier n-Bit Zahlen bekannt. So benötigt z.B. ein Algorithmus, der auf *Schönhage* und *Strassen* zurückgeht, nur $\mathcal{O}(n \log n \log \log n)$ Bit-Operationen.

1.2 "Repeated Squaring" (Effizientes Potenzieren)

In einer Reihe kryptographischer Verfahren ist es von essentieller Wichtigkeit, dass man effizient auch große Potenzen von Gruppenelementen berechnen kann. Das in diesem Abschnitt kurz vorgestellte Verfahren kann universell in einer beliebigen Gruppe G verwendet werden, um x^k für ein Element $x \in G$ und $k \in \mathbb{N}$ zu berechnen. Aus diesem Grund geben wir den Algorithmus zur Berechnung solcher Potenzen auch zunächst in sehr allgemeiner Form an.

Algorithmus 1.5. ("Repeated Squaring") Sei G eine Gruppe, $x \in G$ und $k \in \mathbb{N}$. Ferner sei $k = \sum_{i=0}^{n-1} k_i \cdot 2^i$ die Binärdarstellung der Zahl k mit $k_{n-1} = 1$.

(1) Setze $y := x$.

(2) Für $i = n-2, \ldots, 0$ berechne:

$y := y^2$

Ist $k_i = 1$, so berechne zusätzlich $y := y \cdot x$.

(3) Gebe $y = x^k$ aus.

Anstatt eines Korrektheitsbeweises, bei dem man die Gültigkeit einer entsprechend geschickt gewählten Invariante nachweist, betrachten wir ein kleines Beispiel:

Beispiel 1.6. Sei G eine Gruppe und $x \in G$. Wir wollen x^{20} mit Hilfe von Algorithmus 1.5 berechnen. Es ist $k_4k_3k_2k_1k_0 = 10100$ die Binärdarstellung der Zahl 20. Zuerst setzen wir $y := x$. Wegen $k_3 = 0$ setzen wir $y := y^2 = x^2$ und verzichten auf eine zusätzliche Multiplikation mit x. Im nächsten Schritt ist $k_2 = 1$ zu betrachten. Wir berechnen also zuerst $y := y^2 = x^4$ und dann zusätzlich $y := y \cdot x = x^5$. Wegen $k_1 = k_0 = 0$ sind die nächsten beiden auszuführenden Schritte $y := y^2 = x^{10}$ und zuletzt ist der Wert von y ein weiteres Mal zu quadrieren: $y := y^2 = x^{20}$.

In der Notation von 1.5 erhalten wir für die Laufzeit des Algorithmus:

Korollar 1.7. *Algorithmus 1.5 berechnet in Schritt (2) $\mathcal{O}(n)$-mal das Quadrat des Elementes y und $\mathcal{O}(n)$-mal das Produkt von y mit x. Insgesamt ergibt sich damit eine Laufzeit von $\mathcal{O}(n \cdot M_G)$, wobei M_G die Laufzeit für eine Multiplikation zweier Elemente in G beschreibt.*

1.3 Division mit Rest

Sowohl im Ring der ganzen Zahlen als auch im Polynomring über einem Körper können wir stets *Division mit Rest* durchführen. Division mit Rest ist essentieller Bestandteil des Euklidischen Algorithmus und von modularer Arithmetik. Es soll an dieser Stelle weniger auf algorithmische Details eingegangen werden. Wir betrachten nur einen halbformalen Algorithmus für positive ganze Zahlen, der sich so leicht in einem Computeralgebrasystem programmieren lässt.

Algorithmus 1.8. (Division mit Rest) Seien $a, b \in \mathbb{Z}$, $b > 0$.

(1) Berechne den ganzzahligen Quotienten $q := a \operatorname{div} b := \lfloor \frac{a}{b} \rfloor$ von a und b.

(2) Berechne den Rest $r := a \bmod b := a - q \cdot b$.

(3) Gebe q und r aus.

Dann gilt: $a = q \cdot b + r$ mit $0 \leq r < b$.

Im allgemeinen wird man natürlich normalerweise die Division mit Rest auf den Binärdarstellungen der Zahlen a und b implementieren. Der Einfachheit halber verzichten wir hier auf die Angabe eines solchen Algorithmus. Dass Division mit Rest eine effiziente Rechenoperation ist, besagt das folgende Lemma.

Lemma 1.9. *Sind $a, b \in \mathbb{Z}$ mit Binärlängen n bzw. m, so ist Division mit Rest $a = q \cdot b + r$ mit $q, r \in \mathbb{Z}$ und $0 \leq r < |b|$ mit $\mathcal{O}(m \cdot (n - m))$ Bit-Operationen durchführbar, wobei $n - m$ im wesentlichen die Binärlänge des Quotienten q beschreibt.*

1.4 Der Euklidische Algorithmus

Zur Erinnerung stellen wir diesem Abschnitt die folgenden beiden elementaren Definitionen voran:

Definition 1.10. Sei R ein Ring. Dann heißt a ein Teiler von b, $a \mid b$, falls es ein $r \in R$ gibt mit $b = r \cdot a$.

Definition 1.11. Sei R ein euklidischer Ring (d.h. ein Ring, in dem man zwei Elemente per Division mit Rest durcheinander dividieren kann).

(i) Dann heißt $d \in R$ ein *größter gemeinsamer Teiler* von $a, b \in R$, $d = \mathrm{ggT}(a, b)$, falls $d \mid a$ und $d \mid b$ und für alle $s \in R$ mit $s \mid a$ und $s \mid b$ stets $s \mid d$ folgt.

(ii) Ein Element $k \in R$ heißt *kleinstes gemeinsames Vielfaches* von a und b, $k = \mathrm{kgV}(a, b)$, falls $a \mid k$ und $b \mid k$ und für alle $l \in R$ mit $a \mid l$ und $b \mid l$ stets $k \mid l$ folgt.

Für $R = \mathbb{Z}$ sprechen wir häufig von "dem" größten gemeinsamen Teiler und meinen damit den eindeutig bestimmten positiven größten gemeinsamen Teiler zweier ganzer Zahlen. Gleiches gilt für das kleinste gemeinsame Vielfache zweier ganzer Zahlen. Im folgenden Lemma fassen wir ohne Beweis eine der für uns wesentlichen Eigenschaften des größten gemeinsamen Teilers ganzer Zahlen zusammen:

Lemma 1.12. *Seien $a, b \in \mathbb{Z}$. Dann gilt: Ist $g = \mathrm{ggT}(a, b)$, $g > 0$, so existieren $s, t \in \mathbb{Z}$ mit $g = s \cdot a + t \cdot b$.*

Die Darstellung in Lemma 1.12 liefert der *Erweiterte Euklidische Algorithmus*, der in einem beliebigen euklidischen Ring durchgeführt werden kann. Wir geben ihn der Einfachheit halber nur für ganze Zahlen an. Die hier vorgestellte Version kann aber leicht in eine Version für Polynome "umgeschrieben" werden, wenn man die entsprechenden Ungleichungen als Ungleichungen über den Grad der betrachteten Polynome interpretiert.

Vorab aber noch ein kleines Lemma, das uns beim Beweis der Korrektheit des Algorithmus sehr nützlich sein wird:

Lemma 1.13. *Seien $a, b \in \mathbb{Z}$. Dann gilt:* $\mathrm{ggT}(a, b) = \mathrm{ggT}(a \bmod b, b)$.

Beweis. Division mit Rest liefert $a = q \cdot b + (a \bmod b)$. Es gilt: $\mathrm{ggT}(a, b)$ teilt sowohl a als auch b, also auch $a \bmod b = a - q \cdot b$ und damit $\mathrm{ggT}(a \bmod b, b)$. Umgekehrt teilt $\mathrm{ggT}(a \bmod b, b)$ sowohl $a \bmod b$ als auch b und damit auch $a = q \cdot b + (a \bmod b)$. Wir erhalten also $\mathrm{ggT}(a, b) \mid \mathrm{ggT}(a \bmod b, b)$ und $\mathrm{ggT}(a \bmod b, b) \mid \mathrm{ggT}(a, b)$. Daraus folgt die Behauptung. □

Algorithmus 1.14. (Erweiterter Euklidischer Algorithmus (EEA)) Seien $a, b \in \mathbb{Z}$ mit $|a| \geq |b|$.

(1) Setze $r_0 := a$, $r_1 := b$, $s_0 := 1$, $s_1 := 0$, $t_0 := 0$, $t_1 := 1$ und $i := 1$.

(2) Wiederhole den folgenden Schritt, bis $r_{l+1} = 0$ gilt für ein $l \in \mathbb{N}_0$:

Berechne $q_i := r_{i-1} \operatorname{div} r_i$ als ganzzahligen Quotienten von r_{i-1} und r_i.
$r_{i+1} := r_{i-1} - q_i \cdot r_i$ (d.h. $r_{i+1} = r_{i-1} \bmod r_i$)
$s_{i+1} := s_{i-1} - q_i \cdot s_i$
$t_{i+1} := t_{i-1} - q_i \cdot t_i$
$i := i + 1$

(3) Gebe r_l, s_l und t_l aus.

Dann ist r_l ein größter gemeinsamer Teiler von a und b und es gilt $r_l = s_l \cdot a + t_l \cdot b$.

Beweis. (*Korrektheit*) Der Erweiterte Euklidische Algorithmus 1.14 terminiert, denn es gilt für r_i mit $i \geq 2$ die Ungleichung $0 \leq r_i = r_{i-2} \bmod r_{i-1} < r_{i-1} \leq a$. Damit bilden die Reste r_i eine

streng monoton fallende Folge ganzer Zahlen, die nach unten durch 0 beschränkt ist, d.h. der Algorithmus muss terminieren. Wir zeigen nun, dass für alle i gilt:

$$s_i \cdot a + t_i \cdot b = r_i.$$

Für $i = 0$ und $i = 1$ ist die Behauptung klar per Definition von $r_0, r_1, s_0, s_1, t_0, t_1$. Sei nun $i \geq 2$. Dann folgt per Induktion:

$$\begin{aligned} s_i \cdot a + t_i \cdot b &= (s_{i-2} - q_{i-1} \cdot s_{i-1}) \cdot a + (t_{i-2} - q_{i-1} \cdot t_{i-1}) \cdot b \\ &= (s_{i-2} \cdot a + t_{i-2} \cdot b) - (q_{i-1} \cdot s_{i-1} \cdot a + q_{i-1} \cdot t_{i-1} \cdot b) \\ &= r_{i-2} - q_{i-1} \cdot (s_{i-1} \cdot a + t_{i-1} \cdot b) \\ &= r_{i-2} - q_{i-1} \cdot r_{i-1} \\ &= r_i \end{aligned}$$

Damit folgt für $i = l$ mit $s_l \cdot a + t_l \cdot b = r_l$ die behauptete Darstellung. Es bleibt zu zeigen, dass r_l in der Tat ein größter gemeinsamer Teiler von a und b ist. Mit Lemma 1.13 und $a = (a \operatorname{div} b) \cdot b + (a \bmod b) = q_1 \cdot b + r_2$ folgt $\operatorname{ggT}(a, b) = \operatorname{ggT}(a \bmod b, b) = \operatorname{ggT}(r_2, b) = \operatorname{ggT}(r_2, r_1)$. Wegen $r_{i+1} = r_{i-1} \bmod r_i$ folgt $\operatorname{ggT}(r_{i-1}, r_i) = \operatorname{ggT}(r_{i-1} \bmod r_i, r_i) = \operatorname{ggT}(r_{i+1}, r_i)$. Wegen $r_{l+1} = 0$ und $r_l = \operatorname{ggT}(r_{l+1}, r_l)$ folgt per Rekursion $r_l = \operatorname{ggT}(r_2, r_1) = \operatorname{ggT}(a, b)$. □

Wir betrachten ein Beispiel:

Beispiel 1.15. Seien $a = 126$ und $b = 35$. zunächst definieren wir $r_0 := 126$, $r_1 := 35$, $s_0 := 1$, $s_1 := 0$, $t_0 := 0$ und $t_1 := 1$. Dann folgen wir der Vorschrift aus 1.14:

$$\begin{aligned} i &= 1 \\ q_1 &= r_0 \operatorname{div} r_1 = 126 \operatorname{div} 35 = 3 \\ r_2 &= r_0 - q_1 \cdot r_1 = 126 - 3 \cdot 35 = 21 \\ s_2 &= s_0 - q_1 \cdot s_1 = 1 - 3 \cdot 0 = 1 \\ t_2 &= t_0 - q_1 \cdot t_1 = 0 - 3 \cdot 1 = -3 \end{aligned}$$

$$\begin{aligned} i &= 2 \\ q_2 &= r_1 \operatorname{div} r_2 = 35 \operatorname{div} 21 = 1 \\ r_3 &= r_1 - q_2 \cdot r_2 = 35 - 1 \cdot 21 = 14 \\ s_3 &= s_1 - q_2 \cdot s_2 = 0 - 1 \cdot 1 = -1 \\ t_3 &= t_1 - q_2 \cdot t_2 = 1 - 1 \cdot (-3) = 4 \end{aligned}$$

$$\begin{aligned} i &= 3 \\ q_3 &= r_2 \operatorname{div} r_3 = 21 \operatorname{div} 16 = 1 \\ r_4 &= r_2 - q_3 \cdot r_3 = 21 - 1 \cdot 14 = 7 \\ s_4 &= s_2 - q_3 \cdot s_3 = 1 - 1 \cdot (-1) = 2 \\ t_4 &= t_2 - q_3 \cdot t_3 = -3 - 1 \cdot 4 = -7 \end{aligned}$$

$$\begin{aligned} i &= 4 \\ q_4 &= r_3 \operatorname{div} r_4 = 14 \operatorname{div} 7 = 2 \\ r_5 &= r_3 - q_4 \cdot r_4 = 14 - 2 \cdot 7 = 0 \end{aligned}$$

Damit folgt: $\mathrm{ggT}(126, 35) = r_4 = 7$ und $7 = 2 \cdot 126 + (-7) \cdot 35$.

Nun wollen wir uns überlegen, welche Kosten der Erweiterte Euklidische Algorithmus für zwei ganze Zahlen a und b beansprucht.

Theorem 1.16. *Der Erweiterte Euklidische Algorithmus 1.14 für zwei Zahlen $a, b \in \mathbb{Z}$ mit $|a| \geq |b|$ und $n := \lfloor \log_2 |a| \rfloor + 1$ benötigt $\mathcal{O}(n)$ Wiederholungen von Schritt (2) und insgesamt eine Laufzeit von $\mathcal{O}(n^2)$ Bit-Operationen.*

Beweis. Im folgenden setzen wir o.E. $a, b \geq 0$ voraus. Es sei l die Anzahl der Schritte, die der Erweiterte Euklidische Algorithmus bei Eingabe a und b benötigt, bis er s_l, t_l, r_l ausgibt mit $r_l = \mathrm{ggT}(a, b)$ und $r_l = s_l \cdot a + t_l \cdot b$. Dann erhalten wir in der Notation von 1.14

$$r_{i-1} = q_i \cdot r_i + r_{i+1} \geq r_i + r_{i+1} > 2 \cdot r_{i+1},$$

da die Folge der Reste eine streng monoton fallende Folge nichtnegativer ganzer Zahlen bildet. Also folgt

$$\prod_{i=2}^{l-1} r_{i-1} > 2^{l-2} \cdot \prod_{i=2}^{l-1} r_{i+1}.$$

Mit Hilfe dieser Abschätzung gewinnen wir aber eine Abschätzung für die Anzahl l der vom Erweiterten Euklidischen Algorithmus benötigten Schritte:

$$2^{l-2} < \frac{\prod_{i=2}^{l-1} r_{i-1}}{\prod_{i=2}^{l-1} r_{i+1}} = \frac{r_1 \cdot r_2}{r_{l-1} \cdot r_l} < \frac{r_1^2}{2} \Longrightarrow 2^l < 2 \cdot r_1^2$$

Anwendung von $\log_2$ auf die letzte Ungleichung liefert $l < \log_2(2 \cdot r_1^2) = 1 + 2 \cdot \log_2 r_1$. Wegen $r_1 = b \leq a$ ergibt sich das zentrale Resultat $l \leq 1 + 2 \cdot \log_2 a$ und somit $l \in \mathcal{O}(\log_2 a) = \mathcal{O}(n)$. Den wesentlichen Aufwand in 1.14 machen die Multiplikationen in Schritt (2) aus. Der Aufwand zur Berechnung von $r_{i+1} = r_{i-1} - q_i \cdot r_i$ ist also im wesentlichen durch $\log_2 q_i \cdot \log_2 r_i$ gegeben. Der Gesamtaufwand zur Berechnung aller solcher Werte beläuft sich damit im wesentlichen auf

$$\begin{aligned} \sum_{i=1}^{l} \log_2 q_i \cdot \log_2 r_i &\leq \sum_{i=1}^{l} \log_2 q_i \cdot \log_2 a \\ &= \log_2 a \cdot \sum_{i=1}^{l} \log_2 q_i \\ &= \log_2 a \cdot \log_2 \prod_{i=1}^{l} q_i \\ &\in \log_2 a \cdot \mathcal{O}(\log_2 a) \\ &\in \mathcal{O}((\log_2 a)^2) = \mathcal{O}(n^2) \end{aligned}$$

Man kann zeigen, dass stets gilt $|s_i| \leq \frac{a}{r_{i-1}}$ und $|t_i| \leq \frac{b}{r_{i-1}}$. Es folgt $|s_i| \leq a$ und $|t_i| \leq a$ falls $a > 0$ mit $a > b$. Da für die Werte s_i und t_i also ähnliche Abschätzungen gelten ergibt sich damit eine Gesamtlaufzeit von $\mathcal{O}(n^2)$, denn wir können uns in der obigen Abschätzung für r_i einfach s_i oder auch t_i denken. □

1.5 Modulares Rechnen

Restklassenringe ganzer Zahlen sind für uns **die** algebraische Struktur, in der wir später in kryptographischen Verfahren rechnen wollen. Grundlegend ist die folgende Definition.

Bemerkung und Definition 1.17. Sei $N \in \mathbb{N}$, $N \geq 2$. Auf dem Ring der ganzen Zahlen $\mathbb{Z}$ definieren wir die Relation $\sim_N$ vermöge

$$a \sim_N b :\Longleftrightarrow N \mid a - b.$$

Die Relation $\sim_N$ ist eine Äquivalenzrelation, so dass wir die Äquivalenzklassen nach $\sim_N$ betrachten können. Die Klasse von $a \in \mathbb{Z}$ modulo N definieren wir als

$$a \bmod N := [a]_N := \{b \in \mathbb{Z} \mid b \sim_N a\}.$$

Als Repräsentanten für $a \bmod N$ wählen wir immer die kleinste nicht-negative ganze Zahl z mit $0 \leq z \leq N-1$ aus der Menge $\{b \in \mathbb{Z} : N \mid (b-a)\}$. Die Menge der Äquivalenzklassen nach der Äquivalenzrelation $\sim_N$ bezeichnen wir mit $\mathbb{Z}_N$. Wir definieren für $a_1 \bmod N, a_2 \bmod N \in \mathbb{Z}_N$ eine Addition und eine Multiplikation wie folgt:

$$a_1 \bmod N + a_2 \bmod N = (a_1 + a_2) \bmod N,$$
$$a_1 \bmod N \cdot a_2 \bmod N = (a_1 \cdot a_2) \bmod N.$$

Mit der so definierten Addition und Multiplikation wird $\mathbb{Z}_N$ zu einem kommutativen Ring mit Nullelement $0 \bmod N$ und Einselement $1 \bmod N$, der als *Restklassenring von $\mathbb{Z}$ modulo N* bezeichnet wird. $\mathbb{Z}_N$ hat N Elemente. Ein bezüglich der Multiplikation invertierbares Element $a \bmod N$ bezeichnen wir als *Einheit modulo N*. Die Menge aller Einheiten von $\mathbb{Z}_N$ wird mit $\mathbb{Z}_N^\times$ bezeichnet:

$$\mathbb{Z}_N^\times = \{i \in \mathbb{Z}_n;\ i \text{ ist modulo } N \text{ invertierbar}\}.$$

Beispiel 1.18. Sei $N = 6$. Dann besteht $\mathbb{Z}_N = \mathbb{Z}_6$ aus den Restklassen $0 \bmod 6$, $1 \bmod 6$, $2 \bmod 6$, $3 \bmod 6$, $4 \bmod 6$ und $5 \bmod 6$. Es gilt:

$$2 \bmod 6 + 3 \bmod 6 = (2+3) \bmod 6 = 5 \bmod 6,$$
$$4 \bmod 6 + 5 \bmod 6 = (4+5) \bmod 6 = 9 \bmod 6 = 3 \bmod 6,$$
$$2 \bmod 6 \cdot 2 \bmod 6 = (2 \cdot 2) \bmod 6 = 4 \bmod 6,$$
$$2 \bmod 6 \cdot 3 \bmod 6 = (2 \cdot 3) \bmod 6 = 6 \bmod 6 = 0 \bmod 6,$$
$$5 \bmod 6 \cdot 5 \bmod 6 = (5 \cdot 5) \bmod 6 = 25 \bmod 6 = 1 \bmod 6.$$

Unter anderem im Rahmen des RSA Verfahrens 3.1 sind wir an den Einheiten $\mathbb{Z}_N^\times$ modulo einer Zahl N interessiert. Es gilt:

Lemma 1.19. *Für $N \in \mathbb{N}$ gilt $\mathbb{Z}_N^\times = \{k \bmod N \mid \mathrm{ggT}(k, N) = 1\}$. Mit anderen Worten erhalten wir:*

$$\boxed{k \text{ ist invertierbar modulo } N \text{ genau dann, wenn } \mathrm{ggT}(k, N) = 1.}$$

Beweis. Es sei $r \in \mathbb{Z}_N^\times$. Dann gibt es ein $l \in \mathbb{Z}_N^\times$ mit $r \cdot l \equiv 1 \bmod N$, d.h. $r \cdot l = m \cdot N + 1$ für ein $m \in \mathbb{N}$. Folglich ist $r \cdot l - m \cdot N = 1$ und $\mathrm{ggT}(r, N) \mid 1$. Es folgt $\mathrm{ggT}(r, N) = 1$. Umgekehrt liefert für $r \bmod N \in \{k \bmod N \mid \mathrm{ggT}(k, N) = 1\}$ der Erweiterte Euklidische Algorithmus 1.14 die Darstellung $1 = s \cdot r + t \cdot N$ mit $s, t \in \mathbb{Z}$. Reduktion modulo N liefert $s \bmod N$ als Inverses von $r \bmod N$. □

Der Beweis des obigen Lemmas ist konstruktiv und liefert uns unmittelbar den folgenden Algorithmus zur Berechnung modularer Inverser. In Wirklichkeit ist der folgende Algorithmus ein wenig allgemeiner formuliert. Als Eingabe werden $k, N \in \mathbb{N}$ erwartet. Ausgabe ist dann entweder das modulare Inverse von k modulo N, falls k invertierbar modulo N ist, oder "k ist nicht invertierbar modulo N".

Bevor wir den Algorithmus explizit angeben, betrachten wir noch ein kleines Beispiel:

Beispiel 1.20. Gesucht sei das modulare Inverse von $23 \bmod 110$. Der Erweiterte Euklidische Algorithmus 1.14 liefert uns die Darstellung

$$1 = -43 \cdot 23 + 9 \cdot 110.$$

Reduktion modulo 110 liefert also

$$\begin{aligned} 1 \bmod 110 &\equiv (-43 \bmod 110) \cdot (23 \bmod 110) + (9 \bmod 110) \cdot \underbrace{(110 \bmod 110)}_{=0 \bmod 110} \\ &\equiv (-43 \bmod 110) \cdot (23 \bmod 110) \\ &\equiv (67 \bmod 110) \cdot (23 \bmod 110), \end{aligned}$$

also gilt $(23 \bmod 110)^{-1} \equiv (67 \bmod 110)$.

Algorithmus 1.21. (Berechnung des modularen Inversen) Sei $N \in \mathbb{N}$ und $k \in \mathbb{Z}_N$.

(1) Berechne mit Hilfe des Erweiterten Euklidischen Algorithmus 1.14 die Darstellung $g = \mathrm{ggT}(k, N) = s \cdot k + t \cdot N$.

(2) Ist $g = 1$, so gebe $s \bmod N$ aus. Andernfalls gebe "k ist nicht invertierbar modulo N" aus.

Die Korrektheit des Algorithmus folgt aus Lemma 1.19. Setzen wir $k \leq N$ voraus, so folgt nach Theorem 1.16, dass das modulare Inverse von $k \bmod N$ mit $\mathcal{O}(n^2)$ Bit-Operationen berechnet werden kann, wobei $n = \lfloor \log_2 N \rfloor + 1$. Damit erhalten wir die folgenden Laufzeiten für modulare Arithmetik:

Korollar 1.22. *Sei $N \in \mathbb{N}$ und $n := \lfloor \log_2 N \rfloor + 1$ die Bit-Länge der Zahl N. Die Summe zweier Elemente aus $\mathbb{Z}_N$ läßt sich mit $\mathcal{O}(n)$ Bit-Operationen berechnen. Für die Multiplikation zweier Elemente (nach der klassischen Methode) erhalten wir eine Laufzeit von $\mathcal{O}(n^2)$ Bit-Operationen. Berechnung des Inversen eines Elementes aus $\mathbb{Z}_N^\times$ ist mit $\mathcal{O}(n^2)$ Bit-Operationen durchführbar. Ferner hat Algorithmus 1.5 zum effizienten Potenzieren eine Laufzeit von $\mathcal{O}(n^3)$ (wenn man zur Multiplikation die klassische Methode verwendet und voraussetzt, dass der Exponent $\leq N$ ist).*

Beweis. Die Aussagen über die Laufzeit von Multiplikation und Division sind klar. Nicht so offensichtlich ist die Tatsache, dass die Summe zweier Elemente $k \bmod N$ und $l \bmod N$ tatsächlich nur lineare Laufzeit benötigt, denn wir berechnen zunächst $k + l$ (was sicherlich nur Zeit $\mathcal{O}(n)$ benötigt) und müssen anschließend $k+l$ unter Umständen modulo N reduzieren. Zur Reduktion von $k + l$ modulo N müssen wir aber nicht wirklich Division mit Rest ausführen, denn es gilt $k + l < 2 \cdot N$. Ist also $k + l > N$, so wählen wir schlicht $k + l - N$ als Repräsentanten von $k + l \bmod N$. Dies erfordert nur eine weitere Addition. Die Aussage über die Laufzeit von Algorithmus 1.5 zum effizienten Potenzieren ergibt sich wie folgt: Zur Berechnung von $x^k \bmod N$ können wir (wie wir später sehen werden) ohne Einschränkung $0 \leq k \leq N$ voraussetzen. Damit hat k höchstens eine so große Binärlänge wie die Zahl N selbst. Algorithmus 1.5 führt nun $\mathcal{O}(\log_2 k) = \mathcal{O}(\log_2 N) = \mathcal{O}(n)$ Schleifendurchläufe durch. In jedem Schleifendurchlauf werden entweder eine oder zwei Multiplikationen ausgeführt, die jeweils nach der klassischen Methode eine Laufzeit von $\mathcal{O}(n^2)$ haben. Insgesamt ergibt sich damit die behauptete Laufzeit. □

Als weitere Folgerung aus den bisherigen Ergebnissen erhalten wir:

Korollar 1.23. *Der Ring $\mathbb{Z}_N$ ist ein Körper genau dann, wenn N eine Primzahl ist.*

1.6 Die Eulersche φ-Funktion und einige ihrer Eigenschaften

Wir wollen in diesem Abschnitt die *Eulersche φ-Funktion* einführen und einige der Eigenschaften dieser Funktion beweisen, die für uns im kryptographischen Anwendungskontext von Nutzen sein werden.

Definition 1.24. Sei $N \in \mathbb{N}$. Dann ist die *Eulersche φ-Funktion* definiert durch

$$\varphi(N) := \#\{1 \leq x \leq N \mid \mathrm{ggT}(x, N) = 1\}.$$

Beispiel 1.25. Es ist $\varphi(10) = 4$, denn in der Menge $\{1, \ldots, 10\}$ sind nur die Zahlen 1, 3, 7 und 9 teilerfremd zu 10. Besonders einfach ist die Eulersche φ-Funktion auf Primzahlen auszuwerten. Es ist $\varphi(13) = 12$, denn in der Menge $\{1, \ldots, 13\}$ sind alle Zahlen 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 und 12 teilerfremd zu 13. Das folgende Lemma 1.26 ist also naheliegend.

Lemma 1.26. *Für jede Primzahl $p \in \mathbb{N}$ gilt: $\varphi(p) = p - 1$.*

Beweis. Es gilt $\{1 \leq x \leq p \mid \mathrm{ggT}(x,p) = 1\} = \{1, \ldots, p-1\}$. Dies zeigt die Behauptung. □

Lemma 1.27. *Sei $p \in \mathbb{N}$ eine Primzahl und $e \in \mathbb{N}$. Dann gilt: $\varphi(p^e) = (p-1) \cdot p^{e-1}$.*

Beweis. In der Menge $\{1, \ldots, p^e\}$ sind genau p^{e-1} Zahlen Vielfache von p. Folglich gilt $\varphi(p^e) = p^e - p^{e-1} = (p-1) \cdot p^{e-1}$. □

Die Faktorisierung ganzer Zahlen in ihre Primpotenzfaktoren ist ein schwieriges Kapitel für sich, dem man eine ganze Vorlesungsreihe widmen kann. Wir werden mit Algorithmus 2.9 nur einen der möglichen Algorithmen zur Faktorisierung ganzer Zahlen kennenlernen. Die extrem hohe Komplexität dieses Algorithmus gibt vielleicht eine gewisse Vorstellung dafür, warum der folgende Satz so in dieser Form für uns eher von theoretischem Interesse sein soll und weniger als direkter Algorithmus zur Implementation der Eulerschen φ-Funktion in einem Computeralgebrasystem wie MuPAD dienen kann.

Satz 1.28. *Sei $N = p_1^{e_1} \cdot \ldots \cdot p_r^{e_r}$ die Zerlegung von N in paarweise teilerfremde Primpotenzen. Dann gilt:*

$$\varphi(N) = (p_1 - 1) \cdot p_1^{e_1 - 1} \cdot \ldots \cdot (p_r - 1) \cdot p_r^{e_r - 1}$$

Beweis. Nach Lemma 1.27 gilt: $\varphi(p_i^{e_i}) = (p_i - 1) \cdot p_i^{e_i - 1}$ für $1 \leq i \leq r$. Da die zu N teilerfremden Elemente aus $\{1, \ldots, N\}$ gerade den invertierbaren Elementen modulo N entsprechen, liefert uns der Chinesische Restsatz wegen $\mathbb{Z}_N^\times \cong \mathbb{Z}_{p_1^{e_1}}^\times \times \ldots \times \mathbb{Z}_{p_r^{e_r}}^\times$ die Behauptung, denn $\varphi(N) = \#\mathbb{Z}_N^\times = \prod_{i=1}^r \#\mathbb{Z}_{p_i^{e_i}}^\times = \prod_{i=1}^r \varphi(p_i^{e_i})$. □

1.7 Einige gruppentheoretische Resultate

Wir wollen in diesem Abschnitt die für uns wichtigsten gruppentheoretischen Resultate zusammenfassen. Zur Erinnerung:

Definition 1.29. Die Anzahl der Elemente einer Gruppe G bezeichnen wir stets mit $\#G$ und nennen $\#G$ die *Ordnung* von G. G heißt eine endliche Gruppe, wenn $\#G < \infty$. Sie heißt *zyklisch*, wenn es ein $z \in G$ mit $G = \langle z \rangle$. Dabei bezeichnet $\langle z \rangle$ die Menge $\{z^j \mid j \in \mathbb{Z}\}$ aller Potenzen von z mit ganzzahligen Exponenten.

Für ein beliebiges $x \in G$ ist $\langle x \rangle$ zumindest stets immer eine Untergruppe von G. Diese Untergruppe ist dann im Sinne der obigen Definition eine zyklische Untergruppe von G. Die *Ordnung eines Elementes* $x \in G$, geschrieben $\mathrm{ord}(x)$, ist die Ordnung der von x erzeugten zyklischen Untergruppe $\langle x \rangle$ von G.

1.7.1 Der Satz von Lagrange

Der Satz von Lagrange macht eine wichtige Aussage über die möglichen Ordnungen von Untergruppen einer gegebenen endlichen Gruppe.

Theorem 1.30. (Satz von Lagrange) *Sei G eine endliche Gruppe. Dann gilt:*

(i) Die Ordnung jeder Untergruppe H von G teilt die Ordnung von G, d.h. $\#H \mid \#G$.

(ii) Für jedes Element $x \in G$ gilt: $x^{\#G} = 1$.

Dabei bezeichnet 1 in (ii) das neutrale Element bezüglich der Verknüpfung auf G.

Beweis. (i) Betrachte die *Nebenklassen* $xH := \{x \cdot y \mid y \in H\} \subseteq G$ von H für ein beliebiges $x \in G$. Es gilt $\#(xH) = \#H$, denn die Abbildung $\psi : H \to xH$, $\psi(y) := x \cdot y$ ist bijektiv, denn ψ ist sicherlich surjektiv (per Definition von xH) und aus $\psi(y) = \psi(z)$ folgt $x \cdot y = x \cdot z$, also $y = z$ und damit die Injektivität von ψ. Sind außerdem $x_1, x_2 \in G$, so gilt entweder $x_1H = x_2H$ oder $x_1H \cap x_2H = \varnothing$. Gilt nämlich $x_1 \cdot y = x_2 \cdot z \in x_1H \cap x_2H$, so folgt wegen $x_1 = x_2 \cdot (z \cdot y^{-1})$ gerade $x_1 \in x_2H$. Aus $x_1 \in x_2H$ folgt für jedes $x \in H$ gerade $x_1 \cdot x = x_2 \cdot (z \cdot y^{-1} \cdot x) \in x_2H$. Dies zeigt $x_1H \subseteq x_2H$. Aus Symmetriegründen folgt daraus aber bereits $x_1H = x_2H$. Insgesamt bilden damit die Nebenklassen xH eine Partition von G, denn ein Element $x \in G$ ist stets in der Nebenklasse xH enthalten (da H als Untergruppe von G insbesondere das neutrale Element 1 von G enthalten muss) und G läßt sich damit als disjunkte Vereinigung der Nebenklassen schreiben. Gibt es k solcher verschiedener Nebenklassen, so folgt: $\#G = k \cdot \#H$.
(ii) Betrachte $H := \langle x \rangle$. Wegen $x \in H$ und $x \in xH$ folgt nach Teil (i) bereits $H = xH$. Wir erhalten also wegen $\prod_{y \in H} y = \prod_{y \in xH} y$ gerade

$$x^{\#H} \cdot \prod_{y \in H} y = \prod_{y \in H} x \cdot y = \prod_{y \in H} y \quad \Longrightarrow \quad x^{\#H} = 1$$

und da $\#G$ ein Vielfaches von $\#H$ ist, folgt daraus die Behauptung. □

Korollar 1.31. *Sei G eine Gruppe und $x \in G$ mit $\langle x \rangle$ ist endlich. Dann gilt:*

(i) $\{i \in \mathbb{Z} \mid x^i = 1\}$ ist das von $\operatorname{ord}(x)$ in $\mathbb{Z}$ erzeugte Hauptideal.

(ii) $\operatorname{ord}(x) = \min\{i > 0 \mid x^i = 1\}$

(iii) $\operatorname{ord}(x) \mid \#G$

Beweis. Sei k der kleinste positive Wert mit $x^k = 1$. Ein solches k existiert, da $\langle x \rangle$ endlich ist. Sei $i \in \mathbb{Z}$. Schreibe $i = q \cdot k + r$ mit $0 \leq r < k$. Dann gilt: $x^i = x^{q \cdot k + r} = (x^k)^q \cdot x^r = x^r$. Dies zeigt $\langle x \rangle \subseteq \{1, x, x^2, \ldots, x^{k-1}\}$. Die umgekehrte Inklusion ist klar, d.h. $\langle x \rangle = \{1, x, x^2, \ldots, x^{k-1}\}$. Alle Elemente $1, x, x^2, \ldots, x^{k-1}$ sind verschieden, da aus $x^i = x^j$ mit $i > j$ sofort $x^{i-j} = 1$ und damit ein Widerspruch zur Minimalität von k folgt. Insgesamt erhalten wir also $\operatorname{ord}(x) = \#\langle x \rangle = k$. Für alle $i \in \mathbb{Z}$ folgt damit

$$x^i = 1 \Longleftrightarrow x^{(i \operatorname{div} k) \cdot k + i \bmod k} = 1 \Longleftrightarrow x^{i \bmod k} = 1 \Longleftrightarrow i \bmod k = 0 \Longleftrightarrow k \mid i.$$

Daraus folgen (i) und (ii). Teil (iii) folgt aus dem Satz von Lagrange 1.30. □

1.7.2 Der Satz von Euler und der kleine Satz von Fermat

Wir erinnern uns, dass für $N \in \mathbb{N}$ der Wert der *Eulerschen φ-Funktion* an der Stelle N gerade die Anzahl der zu N teilerfremden Zahlen aus $\{1, \ldots, N\}$ ist. Die in $\mathbb{Z}_N$ bezüglich der Multiplikation invertierbaren Elemente sind genau durch diejenigen Restklassen gegeben, deren Repräsentanten teilerfremd zu N sind. Folglich besteht die Gruppe $\mathbb{Z}_N^\times$ aus genau $\varphi(N)$ Elementen. Mit anderen Worten:

Lemma 1.32. *Für $N \in \mathbb{N}$ gilt: $\#\mathbb{Z}_N^\times = \varphi(N)$.*

Unmittelbar aus diesem Resultat und dem Satz von Lagrange 1.30 folgt das nächste Theorem, das auf Leonard Euler zurückgeht:

Theorem 1.33. (Satz von Euler) *Für jedes $x \in \mathbb{Z}_N^\times$ gilt $x^{\varphi(N)} = 1$.*

Beweis. Die Behautpung folgt sofort aus $\#\mathbb{Z}_N^\times = \varphi(N)$ und dem Satz von Lagrange 1.30 (ii). □

Aus dem Satz von Euler wiederum ergibt sich unmittelbar das nächste Resultat:

Theorem 1.34. (Kleiner Satz von Fermat) *Sei p eine Primzahl. Dann gilt für alle $x \in \mathbb{Z}_p \setminus \{0\}$: $x^{p-1} = 1$.*

Beweis. Es ist $\varphi(p) = p - 1$ nach 1.26. Da p prim ist, gilt $\mathbb{Z}_p^\times = \mathbb{Z}_p \setminus \{0\}$. Nach dem Satz von Euler 1.33 erhalten wir daher für alle $x \neq 0$: $1 = x^{\varphi(p)} = x^{p-1}$. □

Fassen wir die Resultate von 1.30, 1.33 und 1.34 zusammen, so erhalten wir:

Satz 1.35. *Sei G eine endliche Gruppe und $x \in G$. Dann gilt:*

(i) $\mathrm{ord}(x)$ *teilt $\#G$ und für jedes Vielfache m von $\#G$ gilt $x^m = 1$.*

(ii) Sei $N \in \mathbb{N}$ und $x \in \mathbb{Z}_N^\times$. Dann ist $\mathrm{ord}(x)$ *ein Teiler von $\varphi(N)$ und für jedes Vielfache m von $\varphi(N)$ gilt $x^m = 1$.*

(iii) Sei p eine Primzahl und $x \in \mathbb{Z}_p^\times$. Dann ist $\mathrm{ord}(x)$ *ein Teiler von $p - 1$ und für jedes Vielfache m von $p-1$ gilt $x^m = 1$. Ferner gilt in dieser Situation $x^{m+1} = x$. Insbesondere folgt also $x^p = x$ für alle $x \in \mathbb{Z}_p$.*

Bemerkung 1.36. Wir hatten zu Beginn dieses Abschnitts den Begriff der zyklischen Gruppe wiederholt: Eine endliche Gruppe G heißt *zyklisch*, wenn es ein Element g in G gibt mit

$$G = \langle g \rangle = \{g^k \mid k \in \mathbb{Z}\},$$

d.h. *alle Elemente von G sind als Potenzen von g darstellbar*. Das Element g bezeichnet man dann auch als *Generator von G*. Ein Element $z \in G$ ist genau dann ein Generator von G, wenn $\mathrm{ord}(z) = \#G$. Man kann zeigen, dass die multiplikative Gruppe der Einheiten von $\mathbb{Z}$ modulo einer Primzahl p immer zyklisch ist. Wie man einen Generator von $\mathbb{Z}_p^\times$ findet, ist nicht

selbstverständlich, denn nicht jedes Element der Gruppe ist automatisch ein erzeugendes Element. Ein Beispiel ist $\mathbb{Z}_7^\times$. Dort gilt z.B. $(2 \bmod 7)^3 \equiv 1 \bmod 7$, d.h. $2 \bmod 7$ ist kein Generator von $\mathbb{Z}_7^\times$. Dagegen ist $3 \bmod 7$ ein Generator der Gruppe, denn $(3 \bmod 7)^k \neq 1 \bmod 7$ für alle $k = 1, 2, 3, 4, 5$. Trotzdem erzeugt aber auch $2 \bmod 7$ eine zyklische Gruppe. Diese ist dann eine zyklische Untergruppe von $\mathbb{Z}_7^\times$. Für kryptographische Zwecke ist es in der Regel wichtig, in möglichst großen zyklischen Gruppen zu rechnen. Das Auffinden von Generatoren von $\mathbb{Z}_p^\times$ selbst ist also ebenfalls ein wichtiges Problem, das man in der Praxis in den Griff bekommen muss. Wir werden uns im folgenden aber nicht weiter mit diesem Problem beschäftigen. Es sei nur soviel gesagt, dass man im Prinzip gute Resultate in akzeptabler Zeit erzielt, wenn man einfach ein Element aus $\mathbb{Z}_p^\times$ zufällig wählt. Man muss dann allerdings noch Verfahren an der Hand haben, mit denen sich (zumindest mit hoher Wahrscheinlichkeit) sicherstellen lässt, dass das gewählte Element ganz $\mathbb{Z}_p^\times$ oder zumindest eine große Untergruppe von $\mathbb{Z}_p^\times$ erzeugt.

Kapitel 2

Grundlagen aus der Zahlentheorie

In diesem Kapitel wollen wir die Grundlagen aus der Zahlentheorie erarbeiten, die im Anwendungskontext elementarer Kryptographie von Interesse sind. Wie auch schon im Kapitel "Grundlagen aus Algebra und Computeralgebra" werden wir uns auch hier immer wieder für Algorithmen bzw. ihre Komplexität interessieren. Die wesentlichen hier betrachteten Algorithmen sind einerseits Algorithmen zum Auffinden von Primzahlen und andererseits Algorithmen zur Faktorisierung ganzer Zahlen.

2.1 Finden von Primzahlen und Primzahltest

Folgendes grundlegendes Problem wollen wir uns stellen: Finde einen Algorithmus, der bei Eingabe eines Intervalls $[B, 2 \cdot B - 1]$ eine Primzahl aus diesem Intervall liefert. Wir werden keinen deterministischen Algorithmus angeben, sondern lediglich ein Verfahren, das im wesentlichen einfach zufällig eine Zahl aus dem Intervall $[B, 2 \cdot B - 1]$ wählt und anschließend prüft, ob die gewählte Zahl prim ist. Zunächst müssen wir sicherstellen, dass es "genügend" Primzahlen in $[B, 2 \cdot B - 1]$ für $B \gg 0$ gibt. Die notwendige Aussage, die wir benötigen, liefert der sogenannte *Primzahlsatz*:

Theorem 2.1. (Primzahlsatz) *Sei $x \in \mathbb{N}$ und es bezeichne $\pi(x)$ die Anzahl aller Primzahlen in $[0, x]$. Dann gilt:*

$$\pi(x) \approx \frac{x}{\ln x}$$

bzw. genauer

$$\frac{x}{\ln x} \cdot \left(1 + \frac{1}{2 \cdot \ln x}\right) \leq \pi(x) \leq \frac{x}{\ln x} \cdot \left(1 + \frac{3}{2 \cdot \ln x}\right)$$

für $x \geq 59$.

Der Primzahlsatz garantiert uns für hinreichend großes B die Existenz von mindestens $\frac{B}{2 \cdot \ln B}$ vielen Primzahlen im Intervall $[B, 2 \cdot B - 1]$ (siehe [8], Seite 64).
Wir haben oben bereits erwähnt, dass der Algorithmus zum Auffinden von Primzahlen von einer zufällig gewählten Zahl testen soll, ob sie prim ist. Ein möglicher *Primzahltest* ist der *Fermat-Test*:

Algorithmus 2.2. (Fermat-Test) Sei $M \in \mathbb{Z}$ und $t \in \mathbb{N}$ ein "Konfidenzparameter" (t ist im wesentlichen nichts anderes als die Anzahl der Runden, die der Algorithmus durchführt).

(1) Setze $i := 1$.

(2) Solange $i \leq t$ und die Berechnung nicht abgebrochen wurde, wiederhole die folgenden Schritte:

  (i) Wähle $a \in_R \mathbb{Z}_M \setminus \{0\}$.

  (ii) Berechne $g := \mathrm{ggT}(a, M)$. Falls $g \neq 1$, so breche die Berechnung ab und gebe "M ist zusammengesetzt" aus.

  (iii) Berechne $k := a^{M-1} \bmod M$. Falls $k \not\equiv 1 \bmod M$, so breche die Berechnung ab und gebe "M ist zusammengesetzt" aus.

  (iv) Setze $i := i + 1$.

(3) Gebe "M ist wahrscheinlich prim" aus.

Beweis. (*Korrektheit*) Ist M prim, so ist jedes $a \in \mathbb{Z}_M$, $a \neq 0$ teilerfremd zu M. Ferner gilt dann stets $a^{M-1} \equiv 1 \bmod M$, denn dies ist gerade die Aussage von Fermat's Kleinem Satz 1.34. Damit wird der Algorithmus bei Eingabe eine Primzahl stets die Ausgabe "M ist wahrscheinlich prim" liefern. □

In manchen Verfahren wie z.B. dem Algorithmus *Dixon's Zufallsquadrate* ist es nicht nur wichtig eine Primzahl zu finden, sondern alle Primzahlen $p_1, \ldots, p_h$ bis zu einer vorgegebenen oberen Grenze $B \geq 2$ zu bestimmen. Ein möglicher Algorithmus, der diese Aufgabe bewältigt, ist das *Sieb des Erathostenes*:

Algorithmus 2.3. (Sieb des Erathostenes) Sei $B \geq 2$ gegeben.

(1) Erstelle eine B-elementige Liste L mit $L[i] := i$ für $i = 1, \ldots, B$.

(2) Finde noch nicht markierte Werte $L[i] \neq 0$ mit $i > 1$ minimal. Markiere $L[i]$. Falls kein solches existiert, so gehe zu Schritt (4).

(3) Setze $L[k \cdot i] := 0$ für alle $k \geq 2$ mit $k \cdot i \leq B$.

(4) Gebe alle j mit $L[j] \neq 0$ aus.

Die Korrektheit des Algorithmus ist klar. Seine Laufzeit ergibt sich im wesentlichen zu:

$$\sum_{\substack{p \leq B \\ p \text{ prim}}} \frac{B}{p} = B \cdot \sum_{\substack{p \leq B \\ p \text{ prim}}} \frac{1}{p} \in \mathcal{O}(B \cdot \log_2(\log_2 B)).$$

Bei der Laufzeitanalyse haben wir – nicht ganz korrekter Weise – den Aufwand für das Setzen der Listeneinträge auf den Wert Null nicht mitberechnet. Er macht etwa einen Faktor $\log_2 B$ aus.

2.2 Faktorisierung ganzer Zahlen

In diesem Abschnitt werden wir den Algorihtmus *Pollard's ρ-Methode* zur Faktorisierung ganzer Zahlen kennenlernen.
Gegeben sei eine Zahl $N \in \mathbb{N}$, die weder eine Primzahl, noch eine reine Primpotenz sei. Wir betrachten eine Funktion $f : \mathbb{Z}_N \to \mathbb{Z}_N$ und einen "Startwert" $x_0 \in \mathbb{Z}_N$. Dann berechnen wir die Sequenz der $(x_i)_{i\in\mathbb{N}}$ mit $x_i := f(x_{i-1})$. Wir nehmen an, dass f Werte liefert, die $(x_i)_{i\in\mathbb{N}}$ wie eine zufällige Sequenz erscheinen lassen. Sind nun p und q verschiedene Primfaktoren von N, so sagt uns der Chinesische Restsatz, dass eine modulo N zufällig verteilte Sequenz $(x_i)_{i\in\mathbb{N}}$ auch modulo p und modulo q zufällig verteilt ist. Betrachten wir nun die Sequenz $(x_i)_{i\in\mathbb{N}}$ modulo p, so erhalten wir nach spätestens p-Schritten (und in Wirklichkeit erwartungsgemäß um einiges schneller) eine "Kollision", d.h. wir finden k, l mit $x_k \equiv x_{k+l} \bmod p$. Da aber $(x_i)_{i\in\mathbb{N}}$ modulo N, p und q zufällig verteilt ist, ist es recht unwahrscheinlich, dass zu den gleichen Indizes k, l auch eine Kollision modulo q auftritt. Es gilt also mit hoher Wahrscheinlichkeit $x_k \not\equiv x_{k+l} \bmod q$ und damit auch $x_k \not\equiv x_{k+l} \bmod N$. Letzteres bedeutet aber gerade, dass $\mathrm{ggT}(x_k - x_{k+l}, N)$ nichttrivial ist, womit wir einen Teiler von N gefunden hätten.
Es stellen sich zwei Probleme: Als erstes müssen wir die erwartete Laufzeit bis zum Auftauchen einer Kollision abschätzen. Als zweites müssen wir uns überlegen, wie wir den erhöhten Speicherbedarf des obigen Algorithmus minimieren können, denn wir möchten natürlich nach Möglichkeit nicht alle Werte $(x_i)_{i\in\mathbb{N}}$ speichern müssen, bis schließlich eine Kollision auftritt.

Proposition 2.4. (Das "Geburtstagsproblem") *In einer Urne befinden sich p gleichartige Kugeln mit den Nummern $1, \ldots, p$. Dann muss man erwartungsgemäß $\mathcal{O}(\sqrt{p})$ Ziehungen mit Zurücklegen durchführen, bis das erste Mal eine Kollision auftritt (d.h. bis das erste Mal eine Kugel gezogen wird, die schon bei einem der vorhergehenden Züge zum Vorschein gekommen war).*

Beweis. Es sei s die Zufallsgröße, die die Anzahl der benötigten Ziehungen bis zur ersten Kollision beschreibt. Gesucht ist der Erwartungswert $\mathcal{E}(s)$. Es gilt:

$$\begin{aligned}
\mathcal{E}(s) &= \sum_{j=1}^{\infty} j \cdot \mathrm{P}(s = j) = \sum_{j=1}^{\infty} j \cdot (\mathrm{P}(s \geq j) - \mathrm{P}(s \geq j+1)) \\
&= \sum_{j=1}^{\infty} j \cdot \mathrm{P}(s \geq j) - \sum_{j=1}^{\infty} j \cdot \mathrm{P}(s \geq j+1) \\
&= \mathrm{P}(s \geq 1) + \sum_{j=2}^{\infty} j \cdot \mathrm{P}(s \geq j) - \sum_{j=2}^{\infty} (j-1) \cdot \mathrm{P}(s \geq j) \\
&= \mathrm{P}(s \geq 1) + \sum_{j=2}^{\infty} \mathrm{P}(s \geq j) = \sum_{j=1}^{\infty} \mathrm{P}(s \geq j)
\end{aligned}$$

Die Wahrscheinlichkeit in den ersten $j-1$ Ziehungen keine Kollision zu erhalten lässt sich wie

folgt berechnen und weiter abschätzen:

$$\begin{aligned}\mathrm{P}(s \geq j) &= \frac{p \cdot (p-1) \cdot (p-2) \cdot \ldots \cdot (p-j+2)}{p^{j-1}} \\ &= \prod_{1 \leq i < j} \frac{p-(i-1)}{p} = \prod_{1 \leq i < j} \left(1 - \frac{i-1}{p}\right) \\ &\leq \prod_{1 \leq i < j} e^{-(i-1)/p} = e^{-(j-2)\cdot(j-1)/(2\cdot p)} \leq e^{-(j-2)^2/(2\cdot p)},\end{aligned}$$

wobei wir $1-x \leq e^{-x}$ benutzt haben. Damit ergibt sich für den Erwartungswert die Abschätzung:

$$\begin{aligned}\mathcal{E}(s) &\leq \sum_{j=1}^{\infty} e^{-(j-2)^2/(2\cdot p)} \leq 1 + \sum_{j=0}^{\infty} e^{-j^2/(2\cdot p)} \leq 2 + \int_0^{\infty} e^{-x^2/(2\cdot p)} dx \\ &\leq 2 + \sqrt{2\cdot p} \cdot \int_0^{\infty} e^{-x^2} dx = 2 + \sqrt{2\cdot p} \cdot \frac{\sqrt{\pi}}{2} = 2 + \sqrt{\frac{\pi \cdot p}{2}} \\ &\in \mathcal{O}(\sqrt{p})\end{aligned}$$

□

Damit haben wir das erste der beiden Probleme gelöst: Ist p der kleinste Primfaktor der Zahl N, so können wir nach $\mathcal{O}(\sqrt{p})$-Schritten mit einer Kollision rechnen. Ein möglicher Ansatz, die Zahl N zu faktorisieren, könnte also darin bestehen, die Werte $(x_i)_{i\in\mathbb{N}}$ zufällig zu wählen und auf eine Kollision "zu warten". Die Laufzeit dieses Algorithmus entspricht dann im wesentlichen der Anzahl der zu erwartenden "Züge" bis zur ersten Kollision.
Nun zum zweiten Problem: Anstatt alle Werte x_i für $i \geq 1$ mit $x_i = f(x_{i-1})$ bis zur ersten Kollision zu speichern, benutzen wir den sogenannten *Zykel-Trick von Floyd*:

Algorithmus 2.5. (Floyd's Zykel Trick) Sei $x_0 \in \{0, \ldots, p-1\}$ und $f : \{0, \ldots, p-1\} \to \{0, \ldots, p-1\}$ eine Funktion.

(1) Setze $i := 0$ und $y_0 := x_0$.

(2) Wiederhole die folgenden Berechnungen bis $x_i = y_i$ für $i > 0$ gilt:

Setze $i := i+1$, $x_i := f(x_{i-1})$ und $y_i := f(f(y_{i-1}))$.

(3) Gebe i aus.

Beispiel 2.6. Wir betrachten die Funktion $f : \mathbb{Z}_{91}^{\times} \to \mathbb{Z}_{91}^{\times}$, $f(x) := x^2 + 1 \bmod 91$. Sei $x_0 = 5 \bmod 91$. Dann liefert uns Algorithmus 2.5 die beiden Folgen

$$\begin{aligned} x_0 = 5 \bmod 91, \quad x_1 = 26 \bmod 91, \quad x_2 = 40 \bmod 91, \\ x_3 = 54 \bmod 91, \quad \underline{x_4 = 5 \bmod 91}, \end{aligned}$$

$$\begin{aligned} y_0 = 5 \bmod 91, \quad y_1 = 40 \bmod 91, \quad y_2 = 5 \bmod 91, \\ y_3 = 40 \bmod 91, \quad \underline{y_4 = 5 \bmod 91}, \end{aligned}$$

d.h. wir haben bereits nach 4 Schritten eine Kollision modulo 91 gefunden.

Das folgende Lemma zeigt uns, dass eine Kollision nach Floyd's Methode hinreichend schnell auftritt:

Lemma 2.7. *Es sei k der kleinste Wert, für den eine Kollision auftritt, und l die Länge des Zykles, d.h. $x_k = x_{k+l}$. Dann findet Algorithmus 2.5 nach höchstens $(k+l)$-Schritten eine Kollision.*

Beweis. Es gilt $y_i = x_i$ genau dann, wenn $x_{2\cdot i} = x_i$ genau dann, wenn l (dies ist die Länge des Zykels) ein Teiler von $2 \cdot i - i = i$. Der kleinste Wert von i, für den diese Teilbarkeitsbedingung erfüllt ist, ist $i = k+(-k \bmod l) < k+l$ gilt. Letzteres sieht man wie folgt: Es ist $k = (k \operatorname{div} l) \cdot l+(k \bmod l)$, also $-k \bmod l = (k \operatorname{div} l) \cdot l - k$ und damit $i = k+(k \operatorname{div} l) \cdot l - k = (k \operatorname{div} l) \cdot l$. □

Beispiel 2.8. In Beispiel 2.6 hatten wir $f : \mathbb{Z}_{91}^{\times} \to \mathbb{Z}_{91}^{\times}$, $f(x) := x^2 + 1 \bmod 91$ sowie den Startwert $x_0 = 5 \bmod 91$ betrachtet. Die Zykel-Iteration liefert die Folge

$$\begin{aligned} x_0 = 5 \bmod 91, \quad \underline{x_1 = 26 \bmod 91}, \quad x_2 = 40 \bmod 91, \\ x_3 = 54 \bmod 91, \quad x_4 = 5 \bmod 91, \end{aligned}$$

$$\begin{aligned} y_0 = 5 \bmod 91, \quad \underline{y_1 = 40 \bmod 91}, \quad y_2 = 5 \bmod 91, \\ y_3 = 40 \bmod 91, \quad y_4 = 5 \bmod 91. \end{aligned}$$

Wir stellen uns jetzt vor, wir wollten die Zahl 91 faktorisieren. Einer der Primfaktoren ist 7. Schauen wir uns obige Folge von Werten modulo 7 an, so stellt sich eine Kollision modulo 7 viel eher ein als modulo 91:

$$x_0 = 5 \bmod 7, \quad x_1 = 5 \bmod 7$$

$$y_0 = 5 \bmod 7, \quad y_1 = 5 \bmod 7.$$

Wir haben also eine Kollision modulo des kleinsten Primteilers von 91, nicht aber modulo 91. Berechnen wir den größten gemeinsamen Teiler von $40 - 26 = 14$ und 91, so finden wir diesen Pimteiler: $\operatorname{ggT}(14, 91) = 7$.

Algorithmus 2.9. (Pollard's ρ-Methode) Sei $N \in \mathbb{N}$ keine Primzahl.

(1) Wähle $x_0 \in_R \{0, \ldots, N-1\}$ und setze $y_0 := x_0$ sowie $i := 0$.

(2) Wiederhole die folgenden Schritte:

(i) Setze $i := i + 1$, $x_i := x_{i-1}^2 + 1 \bmod N$, $y_i := (y_{i-1}^2 + 1)^2 + 1 \bmod N$.

(ii) Berechne $g := \operatorname{ggT}(x_i - y_i, N)$. Falls $1 < g < N$, so gebe g aus. Falls $g = N$, so breche das Verfahren ab.

Als Laufzeit für den Algorithmus erhalten wir

Theorem 2.10. *In der Notation von Algorithmus 2.9 hat Pollard's ρ-Methode eine Laufzeit von $\mathcal{O}(\sqrt{p} \cdot (\log_2 N)^2)$ Bit-Operationen, wobei p der kleinste Primteiler der Zahl N sei. Wegen $p < \sqrt{N} = N^{1/2}$ läßt sich die Laufzeit auch ausschließlich in Abhängigkeit von N angeben. Die Laufzeit ist dann $\mathcal{O}(N^{1/4} \cdot (\log_2 N)^2)$.*

Beweis. Die Behauptung folgt sofort mit den Laufzeitabschätzungen für Floyd's Zykel Trick, der Laufzeit für den Erweiterten Euklidischen Algorithmus 1.16 sowie dem Aufwand für modulare Multiplikationen. Beachte, dass wir nach $\mathcal{O}(\sqrt{p})$ Schritten mit einer Kollision rechnen können, d.h. der Zykel, in den die berechneten Werte eintreten, kann höchstens Länge $\mathcal{O}(\sqrt{p})$ haben. Damit liefert Floyd's Zykel Trick erwartungsgemäß nach $2 \cdot \sqrt{p}$ Schritten eine Kollision, also in erwarteter Zeit $\mathcal{O}(\sqrt{p})$. □

Kapitel 3

Einige Kryptosysteme

3.1 Vorbemerkungen

In diesem Kapitel werden einige der bekanntesten Kryptosysteme vorgestellt, darunter: *RSA*, das sogenannte *Diffie-Hellman-Schlüsselaustauschverfahren* und das darauf beruhende Kryptosystem von *ElGamal*.

3.1.1 Symmetrische und Asymmetrische Kryptosysteme

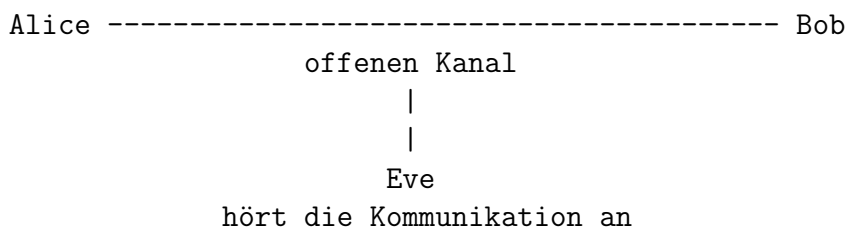
Man unterscheidet grob sogenannte *symmetrische* und *asymmetrische* Kryptosysteme. Symmetrische Kryptosysteme zeichnen sich dadurch aus, dass Sender und Empfänger den gleichen privaten (d.h. geheimen) Schlüssel benutzen, um Nachrichten zu ver- bzw. zu entschlüsseln. Im Gegensatz dazu benötigt in sogenannten asymmetrischen Kryptosystemen (oder auch Public Key Kryptosystemen) nur der Empfänger einen geheimen Schlüssel.
Ein offensichtlicher Nachteil symmetrischer Kryptosysteme ist damit die Tatsache, dass sich die Parteien, die miteinander kommunizieren möchten, auf einen gemeinsamen (geheimen) Schlüssel zum Ver- und Entschlüsseln von Nachrichten einigen müssen. Es muss eine sichere Möglichkeit gefunden werden, einen solchen Schlüssel auszutauschen. Einer der Vorteile symmetrischer Kryptosysteme ist dagegen i.a. ihre hohe Effizienz. So kann z.B. das symmetrische Kryptosystem Rijndael effizient in Hardware implementiert werden (und wurde von seinen Erfindern auch unter diesem Aspekt entwickelt). Ein weiterer Vorteil von symmetrischen Kryptosystemen ist die Tatsache, dass sich der Empfänger einer verschlüsselten Nachricht, wenn er diese mit seinem geheimen Schlüssel entschlüsseln konnte, über die Identität des Absenders weitestgehend im Klaren sein kann (es sei denn, der geheime Schlüssel ist unbemerkt in die Hände böswilliger Dritter gelangt, die ihn nun dazu verwenden, Falschmeldungen oder Fehlinformationen etc. an den Empfänger zu senden).
Bei asymmetrischen Kryptosystemen, in denen jede beliebige Person mit einem öffentlichen Schlüssel Nachrichten an den Empfänger verschlüsseln und an ihn versenden kann, ist der Aspekt der Authentifizierung des Absenders in der Regel nicht gegeben. Ein Beispiel für ein solches asymmetrisches Kryptosystem, in dem eine zusätzliche Authentifizierung notwendig

wird, ist das RSA-Verfahren (siehe Seite 27).
Sowohl symmetrische als auch asymmetrische Kryptosysteme werden in der Praxis verwendet. Häufig geht man so vor, dass man ein asymmetrisches Kryptosystem dazu verwendet, einen gemeinsamen geheimen Schlüssel für ein symmetrisches Kryptosystem sicher unter den kommuzierenden Parteien auszutauschen. Anschließend wird dann die Kommunikation über (effizientere) symmetrische Kryptosysteme geführt.

Wir werden uns aus Zeitgründen nur mit asymmetrischen Kryptosystemen beschäftigen.

3.1.2 Alice, Bob und Eve

Sender und Empfänger werden traditioneller Weise stets mit den Namen Alice und Bob versehen. Alice und Bob kommunizieren über einen offenen Kanal (z.B. das Internet) und Bob möchte Alice eine Nachricht übersenden derart, dass eine dritte, böswillige Person Eve, die den Kanal abhört, die Nachricht nicht verstehen kann.



Die bezeichnende Vorgehensweise ist dann die folgende: Bob möchte Alice eine Nachricht x schicken. Dazu benutzt Bob eine *Verschlüsselungsfunktion* enc_K (Encryption), mit deren Hilfe er ein $y := \mathrm{enc}_K(x)$ erzeugt. K bezeichne dabei den *Schlüssel*, den Bob zum Verschlüsseln der Nachricht benutzt. Dann schickt Bob y an Alice und (vorausgesetzt die Nachricht y wird bei der bloßen Übermittlung nicht verfälscht) Alice kann dann mit Hilfe einer Funktion dec_S (Decryption) aus y den Klartext $x = \mathrm{dec}_S(y)$ berechnen, wobei S ihren privaten Schlüssel zum Entschlüsseln von Nachrichten bezeichnet. Schematisch ergibt sich also der folgende Ablauf:

$$\text{Bob berechnet } y = \mathrm{enc}_K(x) \xrightarrow[\text{Eve hört mit}]{\text{Bob sendet } y} \text{Alice berechnet } x = \mathrm{dec}_S(y)$$

Folgende Voraussetzungen sollte man vernünftigerweise verlangen:

1. Bob kann die Nachricht effizient (d.h. in polynomieller Zeit in der Codierungslänge der Eingabe für seine Verschlüsselungsfunktion dec_K) verschlüsseln

2. Eve kann den Wert von $y = \mathrm{enc}_K(x)$ zwar abhören, sollte jedoch nicht in der Lage sein, ohne Kenntnis der Funktion dec_S (die von Alice geheim gehalten werden muss – genauer gesagt muss Alice ihren privaten Schlüssel S geheim halten) x aus y in polynomieller Zeit berechnen zu können

3. Alice kann mit Hilfe von dec_S effizient aus y den Wert x berechnen

3.2 Das RSA-Verfahren

Das RSA-Kryptosystem ist nach seinen Erfindern *R. L. Rivest*, *A. Shamir* und *L. M. Adleman* benannt. Es handelt sich bei diesem Verfahren um ein asymmetrisches oder Public-Key-Kryptosystem. Der Ablauf des RSA-Verfahrens wird wieder in unserem Model als Kommunikation zwischen Alice und Bob formuliert. Wenn Bob an Alice eine mit RSA verschlüsselte Nachricht, die wir uns als natürliche Zahl vorstellen, schicken möchte gehen beide nach dem folgenden Protokoll vor:

Protokoll 3.1. (Das RSA-Verfahren) Gegeben sei ein Sicherheitsparameter $n \in \mathbb{N}$. Bevor Alice und Bob miteinander kommunizieren können, trifft Alice die folgenden Vorbereitungen:

(1) Alice wählt zufällig zwei verschiedene Primzahlen p und q im Intervall $[\lceil 2^{\frac{n-1}{2}} \rceil, \lfloor 2^{\frac{n}{2}} \rfloor]$.

(2) Sie berechnet $N := p \cdot q$ und $\varphi(N) = (p-1) \cdot (q-1)$.

(3) Dann wählt sie $e \in_R \{2, \ldots, \varphi(N) - 2\}$ mit $\mathrm{ggT}(e, \varphi(N)) = 1$.

(4) Alice bestimmt d mit $e \cdot d \equiv 1 \bmod \varphi(N)$.

(5) Schließlich veröffentlicht sie $K := (N, e)$ als ihren öffentlichen Schlüssel und hält das Paar $S := (N, d)$ geheim.

(6) Die Werte p, q und $\varphi(N)$ löscht Alice.

Angenommen Bob möchte an Alice die Nachricht x mit $x \in \{0, \ldots, N-1\}$ schicken:

(7) Bob berechnet $y := x^e \bmod N$ und schickt y an Alice.

Alice kann dann die Nachricht y wie folgt entschlüsseln:

(8) Alice berechnet $x^\star := y^d \bmod N$.

Dann gilt $x^\star = x$.

Beweis. (*Korrektheit*) Wir müssen $x^\star = x$ zeigen. Ist $x \in \mathbb{Z}_N^\times$, so folgt die Behauptung unmittelbar aus dem Satz von Euler 1.33 und $e \cdot d \equiv 1 \bmod \varphi(N)$, denn ist $e \cdot d - 1 = k \cdot \varphi(N)$ für ein $k \in \mathbb{N}$, so folgt: $x^\star \equiv x^{e \cdot d} \equiv x^{e \cdot d - 1} \cdot x \equiv (x^{\varphi(N)})^k \cdot x \equiv 1 \cdot x \equiv x \bmod N$. Für ein allgemeines $x \in \mathbb{Z}_N$ gilt ebenso $x^{e \cdot d} \equiv x^{e \cdot d - 1} \cdot x \equiv x^{k \cdot \varphi(N)} \cdot x \equiv x^{k \cdot (p-1) \cdot (q-1)} \cdot x \bmod N$. Wir erhalten also $x^{e \cdot d} \equiv (x^{(p-1)})^{k \cdot (q-1)} \cdot x \equiv x \bmod p$ und $x^{e \cdot d} \equiv (x^{(q-1)})^{k \cdot (p-1)} \cdot x \equiv x \bmod q$, wobei die Identitäten für $x \equiv 0 \bmod p$ und $x \equiv 0 \bmod q$ trivialerweise richtig sind und für $x \not\equiv 0 \bmod p$ und $x \not\equiv 0 \bmod q$ aus dem kleinen Satz von Fermat 1.34 folgen. Da p und q teilerfremd sind und beide $x^{e \cdot d} - x$ teilen, folgt auch $N \mid x^{e \cdot d} - x$, also $x^{e \cdot d} - x \equiv 0 \bmod N$ und damit die Behauptung. □

3.3 Ein erster Angriff auf RSA

Eine schlechte Wahl der Primzahlen p und q ist, wenn man q als die nächste Primzahl wählt, die auf p folgt. In diesem Fall ist RSA dann relativ leicht zu brechen. Der folgende Algorithmus zeigt dies:

Algorithmus 3.2. (Ein erster Angriff auf RSA) Sei $N = p \cdot q$ das Produkt der beiden Primzahlen p und q und q sei die nächste auf p folgende Primzahl.

(1) Berechne $n := \lfloor \sqrt{N} \rfloor$.

(2) Wiederhole die folgenden Schritte, bis eine Primzahl gefunden wird:

> Prüfe mit einem Primzahltest, ob n eine Primzahl ist. Falls n keine Primzahl ist, setze $n := n + 1$.

(3) Gebe n aus.

Die Korrektheit des Algorithmus ist klar. Als Primzahltest könnte z.B. Algorithmus 2.2 verwendet werden (in der Praxis gibt es natürlich effizientere Verfahren). Auch wenn q nicht direkt die nächste auf p folgende Primzahl ist, sondern relativ "nahe" auf p folgt, ist das obige Verfahren durchaus praktikabel. Problematisch ist natürlich der Punkt festzustellen, ob eine gegebene Zahl N, die als Produkt zweier Primzahlen bekannt ist, das Produkt zweier "nahe benachbarter" Primzahlen ist oder nicht. Systematischer, aber auch nicht ohne Zusatzinformation auskommend, ist der im folgenden Abschnitt vorgestellte Angriff von Wiener auf RSA.

3.4 Der Angriff von Wiener auf RSA

Ein bekannter Angriff auf das RSA-Verfahren, wenn der private Schlüssel sehr klein gewählt wird, basiert auf der sogenannten *Kettenbruchentwicklung* einer rationalen Zahl. Die Grundlagen zum Verständnis des "Angriffs von Wiener" sollen hier bereitgestellt werden. Die hier gewählte Darstellung der Theorie ist [1] entnommen (siehe Kapitel 2).

Definition 3.3. Ein *Kettenbruch* ist ein Ausdruck der Form

$$[a_0, a_1, \ldots, a_n] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots\, a_{n-1} + \cfrac{1}{a_n}}}}$$

mit $a_0 \in \mathbb{Q}$ und $a_1, \ldots, a_n \in \mathbb{Q} \setminus \{0\}$.

Das folgende Lemma faßt einige offensichtliche Rechenregeln für Kettenbrüche zusammen:

Lemma 3.4. *Seien $a_0 \in \mathbb{Q}$ und $a_1, \ldots, a_n \in \mathbb{Q} \setminus \{0\}$. Dann gelten die folgenden Rechenregeln für Kettenbrüche:*

(i) $[a_0, a_1, \ldots, a_{n-1}, a_n] = [a_0, a_1, \ldots, a_{n-1}, a_n - 1, 1]$

(ii) $[a_0, a_1, \ldots, a_{n-1}, a_n] = [a_0, a_1, \ldots, a_{n-1} + \frac{1}{a_n}]$

(iii) $[a_0, a_1, \ldots, a_{n-1}, a_n] = a_0 + \dfrac{1}{[a_1, \ldots, a_{n-1}, a_n]}$

Im folgenden werden wir nur Kettenbrüche betrachten, für die $a_i \in \mathbb{N}$ gilt für alle $i = 0, \ldots, n$. Auch wenn Lemma 3.4 (i) besagt, dass verschiedene Kettenbrüche die gleiche Zahl darstellen können, so ist diese eine Mehrdeutigkeit auch die einzige Möglichkeit zwei verschiedene Kettenbrüche für ein- und dieselbe rationale Zahl anzugeben vorausgesetzt alle a_i sind natürliche Zahlen. Im folgenden wird diese Mehrdeutigkeit ignoriert.
Die im folgenden Satz über die Existenz einer Kettenbruchentwicklung für rationale Zahlen vorgenommene Einschränkung auf positive rationale Zahlen, ist keine wirkliche Einschränkung. Man muss nur $a_0 \in \mathbb{Z}$ erlauben und es lassen sich auch negative rationale Zahlen als Kettenbrüche darstellen (soll die Kettenbruchentwicklung von $-q$ für eine positive rationale Zahl q berechnet werden, so betrachte $-\lceil q \rceil + (\lceil q \rceil - q)$, setze $a_0 := -\lceil q \rceil$ und bestimme die Kettenbruchentwicklung der positiven rationalen Zahl $\lceil q \rceil - q$).

Satz 3.5. *Jede positive rationale Zahl $\frac{m_0}{m_1}$ besitzt eine Kettenbruchentwicklung, d.h. es gibt Elemente $a_0, \ldots, a_n \in \mathbb{N}$ mit $[a_0, \ldots, a_n] = \frac{m_0}{m_1}$.*

Der Beweis ist konstruktiv und liefert uns zugleich einen Algorithmus zur Berechnung der Kettenbruchentwicklung einer positiven rationalen Zahl mit Hilfe des Euklidischen Algorithmus 1.14. Wir benötigen in diesem Fall allerdings nicht einmal die Werte s_i und t_i die in 1.14 automatisch mitberechnet werden. Sie müssen für die Korrektheit des folgenden Algorithmus nicht mitberechnet werden.

Algorithmus 3.6. (Kettenbruchentwicklung einer rationalen Zahl) Sei $\frac{m_0}{m_1}$ eine positive rationale Zahl.

(1) Wende den Euklidischen Algorithmus 1.14 auf die Eingabe $a = m_0$ und $b = m_1$ an.

(2) In der Notation von 1.14 wähle $a_0, \ldots, a_n$ als Folge der auftretenden (nichtnegativen ganzzahligen) Quotienten $q_1, \ldots, q_n, q_{n+1}$.

(3) Gebe $[a_0, \ldots, a_n]$ aus.

Beweis. (*Korrektheit*) Der Euklidische Algorithmus liefert uns mit $a_i := q_{i+1}$ die Darstellungen

$$\begin{aligned}
m_0 &= a_0 \cdot m_1 + m_2 && 0 < m_2 < m_1, a_0 \geq 0 \\
m_1 &= a_1 \cdot m_2 + m_3 && 0 < m_3 < m_2, a_1 > 0 \\
&\vdots \quad \vdots && \vdots \\
m_{i-1} &= a_{i-1} \cdot m_i + m_{i+1} && 0 < m_{i+1} < m_i, a_{i-1} > 0 \\
&\vdots \quad \vdots && \vdots \\
m_n &= a_n \cdot m_{n+1} && a_n > 0
\end{aligned}$$

Wir führen den Korrektheitsbeweis per Induktion über n (Anzahl der vom Euklidischen Algorithmus benötigten Schritte). Ist $n = 1$, so gilt $m_0 = a_0 \cdot m_1 + m_2$ und $m_1 = a_1 \cdot m_2$. Damit erhalten wir

$$a_0 + \frac{1}{a_1} = \frac{m_0 - m_2}{m_1} + \frac{1}{\frac{m_1}{m_2}} = \frac{m_0 - m_2}{m_1} + \frac{m_2}{m_1} = \frac{m_0}{m_1},$$

also die Behauptung. Wegen $m_0 = a_0 \cdot m_1 + m_2$ folgt

$$\frac{m_0}{m_1} = a_0 + \frac{m_2}{m_1} = a_0 + \frac{1}{\frac{m_1}{m_2}} = a_0 + \frac{1}{[a_1, \ldots, a_n]} = [a_0, a_1, \ldots, a_n],$$

wobei wir bei der letzten Identität die Induktionsvoraussetzung auf $\frac{m_1}{m_2}$ anwenden konnten. □

Mit Hilfe der Laufzeitabschätzung für den Erweiterten Euklidischen Algorithmus in Theorem 1.16 folgt sofort:

Korollar 3.7. *Ist $\frac{m_0}{m_1}$ eine positive rationale Zahl, so berechnet Algorithmus 3.6 mit einem Aufwand von $\mathcal{O}(n^2)$ Bit-Operationen die Kettenbruchentwicklung von $\frac{m_0}{m_1}$, wobei n das Maximum der Binärlängen von m_0 und m_1 ist.*

Definition 3.8. Sei $[a_0, \ldots, a_n]$ die Kettenbruchentwicklung der positiven rationalen Zahl $\frac{m_0}{m_1}$. Dann heißen die rationalen Zahlen $\frac{p_l}{q_l} = [a_0, \ldots, a_l]$, $l = 0, \ldots, n$ die *Konvergenten* der Kettenbruchentwicklung von $\frac{m_0}{m_1}$.

Für die Konvergenten der Kettenbruchentwicklung einer positiven rationalen Zahl können wir die folgenden Rekursionsformeln angeben, die ihre Berechnung effizient möglich machen:

Lemma 3.9. *Für die Konvergenten $\frac{p_l}{q_l} = [a_0, \ldots, a_l]$, $l = 0, \ldots, n$ einer Kettenbruchentwicklung von $\frac{m_0}{m_1}$ gelten:*

$$\begin{aligned}
p_0 &= a_0 && q_0 = 1 \\
p_1 &= 1 + a_1 \cdot p_0 && q_1 = a_1 \\
p_l &= p_{l-2} + a_l \cdot p_{l-1} && q_l = q_{l-2} + a_l \cdot q_{l-1}, \quad l \geq 2.
\end{aligned}$$

Beweis. Wir führen den Beweis per Induktion nach l. Für $l = 0$ und $l = 1$ rechnen wir die Formeln explizit nach. Für $l = 0$ erhalten wir $\frac{p_0}{q_0} = [a_0] = a_0$, also $p_0 = a_0$ und $q_0 = 1$. Für $l = 1$ gilt $\frac{p_1}{q_1} = [a_0, a_1] = a_0 + \frac{1}{a_1} = \frac{a_1 \cdot a_0 + 1}{a_1}$, also $p_1 = a_1 \cdot a_0 + 1 = a_1 \cdot p_0 + 1$ sowie $q_1 = a_1$. Nach Lemma 3.4 (ii) gilt $\frac{p_{l+1}}{q_{l+1}} = [a_0, \ldots, a_l, a_{l+1}] = [a_0, \ldots, a_l + \frac{1}{a_{l+1}}]$. Nach Induktionsvoraussetzung erhalten wir

$$\left[a_0, \ldots, a_l + \frac{1}{a_{l+1}}\right] = \frac{\left(a_l + \frac{1}{a_{l+1}}\right) \cdot p_{l-1} + p_{l-2}}{\left(a_l + \frac{1}{a_{l+1}}\right) \cdot q_{l-1} + q_{l-2}}.$$

Damit folgt dann:

$$\begin{aligned}
\frac{p_{l+1}}{q_{l+1}} = [a_0, \ldots, a_l, a_{l+1}] &= \left[a_0, \ldots, a_l + \frac{1}{a_{l+1}}\right] \\
&= \frac{\left(a_l + \frac{1}{a_{l+1}}\right) \cdot p_{l-1} + p_{l-2}}{\left(a_l + \frac{1}{a_{l+1}}\right) \cdot q_{l-1} + q_{l-2}} = \frac{a_{l+1} \cdot (\overbrace{a_l \cdot p_{l-1} + p_{l-2}}^{=p_l}) + p_{l-1}}{a_{l+1} \cdot (\underbrace{a_l \cdot q_{l-1} + q_{l-2}}_{=q_l}) + q_{l-1}} \\
&= \frac{a_{l+1} \cdot p_l + p_{l-1}}{a_{l+1} \cdot q_l + q_{l-1}}
\end{aligned}$$

□

Das folgende Lemma geben wir ohne Beweis an:

Lemma 3.10. *Werden die natürlichen Zahlen p_l und q_l wie in Lemma 3.9 berechnet, so sind sie teilerfremd.*

Auf dem folgenden Theorem beruht der "Angriff von Wiener" auf RSA für kleine private Schlüssel. Wir geben ihn hier ohne Beweis an. Ein Beweis findet sich in [6], Lemma 5.7.6, Seite 275.

Theorem 3.11. *Sind $\frac{m_0}{m_1}$ und $\frac{p}{q}$ rationale Zahlen mit*

$$\left|\frac{m_0}{m_1} - \frac{p}{q}\right| < \frac{1}{2 \cdot q^2},$$

so ist $\frac{p}{q}$ eine Konvergente der Kettenbruchentwicklung von $\frac{m_0}{m_1}$.

In Protokoll 3.1 zum RSA-Verfahren ist Alice in Schritt (3) so vorgegangen, dass sie zunächst den öffentlichen Exponenten $e \in_R \{2, \ldots, \varphi(N) - 2\}$ gewählt und dann in Schritt (4) den privaten Exponent d als modulares Inverses von e modulo $\varphi(N)$ bestimmt hat. Bei einer zufälligen Wahl von e auf diese Weise, werden sowohl e als auch d erwartungsgemäß sehr große Zahlen sein.
Bob, der Nachrichten verschlüsseln und an Alice verschicken möchte, benutzt dann den öffentlichen Schlüssel e, während Alice die empfangenen Nachrichten mit Hilfe von d entschlüsselt. Stellen wir uns vor, Alice wäre nichts weiter als eine personifizierte Smart-Card mit sehr geringer Rechenkapazität. Dann könnte Alice auf die Idee kommen, zuerst ein sehr kleines d zu

wählen und dann den Wert von e entsprechend als modulares Inverses von d modulo $\varphi(N)$. Auf diese Weise hätte sich Bob mit einem potentiell sehr großen Exponenten e zum Verschlüsseln einer Nachricht "abzuquälen" und Alice hätte mit einem kleinen Wert für d nur vergleichsweise wenige Operationen zum Entschlüsseln einer Nachricht auszuführen.
Geht Alice also auf die beschriebene Weise vor und wählt sie $d < \frac{1}{3}N^{\frac{1}{4}}$, so werden wir im folgenden zeigen, dass Eve mit Hilfe der erarbeiteten Theorie zu Kettenbrüchen den privaten Schlüssel d von Alice in polynomieller Zeit berechnen kann.

Lemma 3.12. *Es sei $N = p \cdot q$, wobei p, q Primzahlen seien mit $q < p < 2 \cdot q$. Weiter seien $e, d \in \mathbb{Z}_{\varphi(N)}$ mit $e \cdot d \equiv 1 \bmod \varphi(N)$ und $k \in \mathbb{N}$ mit $e \cdot d - 1 = k \cdot \varphi(N)$. Schließlich sei $d < \frac{1}{3} \cdot N^{\frac{1}{4}}$. Dann gilt: $\left|\frac{e}{N} - \frac{k}{d}\right| < \frac{1}{2 \cdot d^2}$.*

Beweis. Es gilt $N - \varphi(N) = p \cdot q - (p-1) \cdot (q-1) = p + q - 1$. Es gilt $q^2 < p \cdot q = N$, also $q < \sqrt{N}$ und $p < 2 \cdot q < 2 \cdot \sqrt{N}$. Damit folgt: $N - \varphi(N) < 3 \cdot \sqrt{N}$. Nun gilt

$$\begin{aligned}\left|\frac{e}{N} - \frac{k}{d}\right| &= \left|\frac{e \cdot d - k \cdot N}{N \cdot d}\right| = \left|\frac{\overbrace{e \cdot d - k \cdot \varphi(N)}^{=1} - k \cdot N + k \cdot \varphi(N)}{N \cdot d}\right| \\ &= \left|\frac{1 - k \cdot (N - \varphi(N))}{N \cdot d}\right| < \frac{3 \cdot k \cdot \sqrt{N}}{N \cdot d} = \frac{3 \cdot k}{d \cdot \sqrt{N}}\end{aligned}$$

Wegen $k \cdot \varphi(N) = e \cdot d - 1 < e \cdot d$ und $e < \varphi(N)$ folgt $k < d$, also $k < \frac{1}{3} \cdot N^{\frac{1}{4}}$. Wir erhalten:

$$\left|\frac{e}{N} - \frac{k}{d}\right| < \frac{N^{\frac{1}{4}}}{d \cdot \sqrt{N}} = \frac{1}{d \cdot N^{\frac{1}{4}}}$$

und wegen $N^{\frac{1}{4}} > 3 \cdot d$ gerade

$$\frac{1}{d \cdot N^{\frac{1}{4}}} < \frac{1}{3 \cdot d^2} < \frac{1}{2 \cdot d^2},$$

womit die Behauptung bewiesen ist. □

Wegen $e \cdot d - k \cdot \varphi(N) = 1$ folgt, dass $\mathrm{ggT}(k, d) = 1$. Theorem 3.11 liefert nun in der Situation des obigen Satzes, dass $\frac{k}{d}$ eine Konvergente der Kettenbruchentwicklung von $\frac{e}{N}$ ist. Da die Konvergenten sowohl effizient berechnet werden können und es nur $\mathcal{O}(\log_2(N))$ viele solcher Konvergenten von $\frac{e}{N}$ gibt (letzteres folgt aus der Tatsache, dass der Erweiterte Euklidische Algorithmus 1.14 bei Eingabe e, N nach Theorem 1.16 nur $\mathcal{O}(\log_2 N)$ Wiederholungen von Schritt (2) benötigt), kann Eve damit durch Ausprobieren aller Konvergenten in polynomieller Zeit den privaten Schlüssel d von Alice berechnen.
Eve kann sogar noch mehr als nur d zu berechnen: Sie kann die Zahl N in polynomieller Zeit faktorisieren. Betrachte dazu das Polynom $f(x) := x^2 - (p+q) \cdot x + p \cdot q$. Dieses Polynom besitzt genau zwei Nullstellen, nämlich $x_1 = p$ und $x_2 = q$. Mit Hilfe der aus der Schule bekannten Formel findet sich eine geschlossene Darstellung dieser beiden Lösungen über

$$x_{1/2} = \frac{p+q}{2} \pm \sqrt{\frac{(p+q)^2}{4} - p \cdot q} = \frac{p+q}{2} \pm \sqrt{\frac{(p+q)^2}{4} - N} \qquad (*)$$

Mit anderen Worten: Kennt Eve den Wert $p+q$, so kann sie in polynomieller Zeit die Nullstellen von $f(x)$, d.h. p und q, berechnen und die Zahl N faktorisieren. Eine Darstellung für die Summe $p+q$ findet sie jedoch in der obigen Situation sehr leicht: Wegen $e \cdot d - 1 = k \cdot \varphi(N)$ kennt sie $\varphi(N) = \frac{e \cdot d - 1}{k}$. Wegen $\varphi(N) = (p-1) \cdot (q-1) = p \cdot q - p - q + 1 = N - (p+q) + 1$ erhält sie $p + q = N - \varphi(N) + 1$. Eve ist $N - \varphi(N) + 1$ in der Situation von oben sehr wohl bekannt, weshalb sie durch Einsetzen des Wertes in $(*)$ die Primfaktoren p und q von N leicht bestimmen kann.

Algorithmus 3.13. (Wiener Angriff auf RSA) Seien $N = p \cdot q$, $q < p < 2 \cdot q$ und e wie im RSA-Verfahren 3.1 gegeben und es gelte zusätzlich für den privaten Schlüssel $d < \frac{1}{3} \cdot N^{\frac{1}{4}}$.

(1) Verschlüssele eine zufällig gewählte Zahl $x \in_R \mathbb{Z}_N \setminus \{0 \bmod N, 1 \mod N\}$ mit Hilfe von e und N zu $\widetilde{x} := x^e \bmod N$.

(2) Berechne mit Hilfe von Algorithmus 3.6 sowie Lemma 3.9 die Konvergenten der Kettenbruchentwicklung von $\frac{e}{N}$.

(3) Für jede der berechneten Konvergenten $\frac{a}{b}$ prüfe, ob $x = \widetilde{x}^b \bmod N$. Ist dies der Fall, so ist b der private Schlüssel des RSA-Verfahrens.

(4) Berechne dann $\varphi(N) = \frac{e \cdot b - 1}{a}$.

(5) Berechne $p + q = N - \varphi(N) + 1$.

(6) Berechne die Primfaktoren p und q von N über

$$x_{1/2} = \frac{p+q}{2} \pm \sqrt{\frac{(p+q)^2}{4} - p \cdot q} = \frac{p+q}{2} \pm \sqrt{\frac{(p+q)^2}{4} - N}.$$

(7) Gebe b (privater Schlüssel) sowie p und q als Faktorisierung von N aus.

3.5 Das Diffie-Hellman-Schlüsselaustauschverfahren

Um einen Schlüsselaustausch für ein Kryptosystem zu ermöglichen, kann man sich z.B. des hier diskutierten Protokolls von *W. Diffie* und *M. E. Hellman* bedienen. Für die im folgenden ganz allgemein gehaltene endliche zyklische Gruppe $G = \langle g \rangle$ der Ordnung $d = \#G$ sind diverse "Kandidaten" denkbar: Die zyklische multiplikative Gruppe $\mathbb{Z}_p^\times$ der Einheiten modulo einer Primzahl p, die zyklische multiplikative Gruppe eines beliebigen Körpers $\mathbb{F}_q^\times$ oder etwa zyklische Untergruppen *elliptischer Kurven*.

Protokoll 3.14. (Das Schlüsselaustauschverfahren von Diffie & Hellman) Sei $G = \langle g \rangle$ ein endliche zyklische Gruppe der Ordnung $d = \#G$. Alice und Bob erhalten dann wie folgt einen gemeinsamen Schlüssel:

(1) Alice wählt ein $s_A \in_R \{2, \ldots, d-1\}$ und sendet $x := g^{s_A}$ an Bob.

(2) Bob wählt ebenfalls ein $s_B \in_R \{2, \ldots, d-1\}$ und sendet $y := g^{s_B}$ an Alice.

(3) Bei Erhalt von y berechnet Alice den gemeinsamen Schlüssel $K_A := y^{s_A}$.

(4) Bei Erhalt von x berechnet Bob den gemeinsamen Schlüssel $K_B := x^{s_B}$.

Dann teilen sich Alice und Bob den gemeinsamen geheimen Schlüssel $K_A = K_B$.

Beweis. (*Korrektheit*) Es gilt $K_A = y^{s_A} = (g^{s_B})^{s_A} = g^{s_A \cdot s_B} = (g^{s_A})^{s_B} = x^{s_B} = K_B$. □

Die Effizienz des Verfahrens ist offensichtlich: Potenzieren erledigen Alice und Bob mit Hilfe von "Repeated Squaring" 1.5. Damit sind alle Operationen in polynomieller Zeit – und damit hinreichend effizient – durchführbar.

3.6 Angriff auf das Verfahren von Diffie & Hellman

In diesem Kapitel wollen wir uns mit diskreten Logarithmen und ihrer Bedeutung in der Kryptographie auseinandersetzen. Wir werden einige Verfahren diskutieren, wie man diskrete Logarithmen von Elementen einer zyklischen (Unter-)Gruppe berechnen kann und ihre Laufzeit analysieren.

3.6.1 Diskrete Logarithmen

Wir betrachten stets eine zyklische Gruppe $G = \langle g \rangle$ mit erzeugendem Element $g \in G$. Im allgemeinen muss man sich nicht von vorneherein auf zyklische Gruppen einschränken, sondern kann ganz beliebige Gruppen wie z.B. elliptische Kurven betrachten und dann die Berechnungen in einer geeigneten zyklischen Untergruppe der betreffenden Gruppe ausführen.

Definition 3.15. Sei $x \in G = \langle g \rangle$ und $\#G = d$. Dann heißt die eindeutig bestimmte Zahl $k \in \{0, \ldots, d-1\}$ mit $x = g^k$ der *diskrete Logarithmus von x in Basis g*. Wir schreiben $k = \mathrm{dlog}_g\, x$.

Gilt $x = g^k$ und ist $d = \#G$, so ist für jedes $i \in \mathbb{Z}$ auch $k + i \cdot d$ ein diskreter Logarithmus von x in Basis g, denn $g^{k+i \cdot d} = g^k \cdot (g^d)^i = g^k \cdot 1^i = g^k = x$. Verlangen wir jedoch $k \in \{0, \ldots, d-1\}$ wie in der obigen Definition, so können wir von *dem* diskreten Logarithmus sprechen: Sind $k, l \in \{0, \ldots, d-1\}$ mit $g^k = x = g^l$, so folgt $g^{k-l} = 1$, also $d \mid k - l$. Wegen $k, l \in \{0, \ldots, d-1\}$ folgt $|k-l| < d$, also wegen $d \mid k-l$ bereits $|k-l| = 0$, d.h. $k = l$.

Das *diskrete Logarithmus Problem* besteht nun darin, zu einem gegeben $x \in G$ dasjenige $k \in \{0, \ldots, d-1\}$ zu finden, für das $x = g^k$ gilt. Beim *allgemeinen diskreten Logarithmus Problem* setzt man nicht zusätzlich voraus, dass $x \in \langle g \rangle$, d.h. bei der allgemeinen Version des Problems ist zusätzlich zu entscheiden, ob das gegebene Element überhaupt ein Element der betrachteten zyklischen Gruppe ist (was die Angelegenheit zusätzlich verkompliziert). Wir werden uns hier ausschließlich mit der spezielleren Variante beschäftigen, bei der wir davon ausgehen können, dass ein gegebenes Element bereits in der zugrundeliegenden zyklischen Gruppe

enthalten ist.

Das diskrete Logarithmus Problem wird als "schwer zu lösen" angesehen. Was genau mit dem Adjektiv "schwer" gemeint ist, werden wir sehen, wenn wir die Komplexität von Algorithmen zur Berechnung des diskreten Logarithmus untersuchen. Ist $n := \lfloor \log_2 d \rfloor + 1$ die Binärlänge der Gruppenordnung, so gilt $d \approx 2^n$. Die Gruppe G hat also 2^n Elemente. Damit liefert der naive Ansatz über ausprobieren aller möglichen Exponenten für g bis man schließlich den Wert x erhält den diskreten Logathmus von x in Basis g nach erwartungsgemäß $\frac{2^n}{2} = 2^{n-1}$ Schritten, was für großes n einen völlig utopischen Rechenaufwand bedeutet.

3.6.2 Der "Baby-Schritt-Riesen-Schritt" Algorithmus

Sei wieder $G = \langle g \rangle$ eine zyklische Gruppe der Ordnung d. Der folgende Algorithmus berechnet den diskreten Logarithmus eines Elementes $x \in G$.

Algorithmus 3.16. (Der "Baby-Schritt-Riesen-Schritt" Algorithmus) Sei $x \in G$.

(1) Setze $m := \lceil \sqrt{d} \rceil$.

(2) "Baby-Schritte": Berechne und speichere $x, x \cdot g, x \cdot g^2, \ldots, x \cdot g^{m-1}$.

(3) "Riesen-Schritte": Berechne $g^m, g^{2 \cdot m}, g^{3 \cdot m}, \ldots$ bis ein $g^{i \cdot m}$ gefunden wird mit $g^{i \cdot m} = x \cdot g^j$.

(4) Gebe $(i \cdot m - j) \bmod d$ aus.

Beweis. (*Korrektheit*) Wir müssen zunächst zeigen, dass es in der Tat überhaupt zu einer Kollision kommt. Im Falle einer Kollision liefert der Algorithmus sicherlich den diskreten Logarithmus von x in Basis g. Es gelte $x = g^a$ für $a \in \{0, \ldots, d-1\}$, $d = \#G$, d.h. $a = \mathrm{dlog}_g\, x$. Division mit Rest liefert $a = b \cdot m + c$ für ein c mit $0 \leq c \leq m-1$. Für $j = m - c$ und $i = b + 1$ gilt $i \cdot m - j = (b+1) \cdot m - (m-c) = b \cdot m + c$. Ist $c = 0$, so folgt $a = b \cdot m$ und $x = g^a = g^{b \cdot m}$. Das Element $g^{b \cdot m}$ wird aber vom Algorithmus berechnet und löst eine Kollision mit x aus. Ist $c > 0$, so folgt $j = m - c \leq m - 1$. Folglich wird $x \cdot g^j$ bei den Baby-Schritten berechnet und löst eine Kollision mit dem Element $g^{i \cdot m}$ für $i = b + 1$ aus, das bei den Riesen-Schritten in jedem Fall berechnet wird. Damit erhalten wir stets die gewünschte Kollision, was die Korrektheit des Algorithmus beweist. □

Theorem 3.17. *Sei $G = \langle g \rangle$ und $d = \#G$. Dann berechnet Algorithmus 3.16 den diskreten Logarithmus in einer erwarteten Anzahl von $\mathcal{O}(\sqrt{d})$ Gruppenoperationen.*

Beweis. In der Notation des obigen Beweises erhalten wir: $b = \frac{a-c}{m} \leq \frac{a}{m} < \frac{d}{m} \leq m$. Die Baby-Schritte benötigen $m-1$ Gruppenoperationen. Bis zu einer Kollision werden nun wegen $i = b+1$ höchstens m Riesen-Schritt Berechnungen ausgeführt. Dies liefert einen Gesamtaufwand von $\mathcal{O}(m)$ Gruppenoperationen, woraus wegen $m = \lceil \sqrt{d} \rceil$ die Behauptung folgt. □

Bemerkung 3.18. Das Speichern aller in den Baby-Schritten berechneten Werte macht Algorithmus 3.16 zu einem sehr speicherintensiven Verfahren. Wir werden in einem der nächsten Abschnitte ein Verfahren mit ähnlicher Laufzeit aber konstantem Speicherplatz (unabhängig von der Gruppenordnung) kennenlernen.

3.6.3 Der Geburtstagsangriff auf den diskreten Logarithmus

Der folgende Algorithmus ist im wesentlichen eine probabilistische Alternative zum Baby-Schritt-Riesen-Schritt Algorithmus 3.16:

Algorithmus 3.19. (Geburtstagsangriff auf DLP) Sei wieder $G = \langle g \rangle$, $d = \#G$ und $x \in G$.

(1) Setze $X := \varnothing$, $Y := \varnothing$.

(2) Wiederhole den folgenden Schritt solange, bis eine Kollision zwischen X und Y auftaucht:

Wähle $i \in_R \{0, \ldots, d-1\}$ und setze $X := X \cup \{x \cdot g^i\}$ sowie $Y := Y \cup \{g^i\}$.

(3) Gilt $x \cdot g^i = g^j$, so gebe $(j - i) \bmod d$ aus.

Beweis. (*Korrektheit*) Da G endlich ist, muss nach endlich vielen Schritten eine Kollision auftreten. Im Falle einer solchen Kollision ist $(j-i) \bmod d$ der gesuchte diskrete Logarithmus. □

Theorem 3.20. *Algorithmus 3.19 berechnet mit einer erwarteten Anzahl von $\mathcal{O}(\sqrt{d})$ Gruppenoperationen den diskreten Logarithmus, $d = \#G$.*

Beweis. Wegen 2.4 können wir nach $\mathcal{O}(\sqrt{d})$ Durchführungen von Schritt (2) mit der ersten Kollision rechnen. Dies zeigt die Behauptung. □

3.6.4 Die Pollard ρ-Methode für den diskreten Logarithmus

Auch Pollard's ρ-Methode zur Berechnung des diskreten Logarithmus basiert darauf, explizit Kollisionen zu produzieren und mit ihrer Hilfe den diskreten Logarithmus zu berechnen. Ziel ist es, den enormen Speicherbedarf der Algorithmen 3.16 und 3.19 auf ein konstantes Minimum zu beschränken. Wie in den vorherigen Abschnitten sei wieder $G = \langle g \rangle$ eine zyklische Gruppe der Ordnung d.

Bemerkung 3.21. Sei $x \in G$ mit $g^a = x$. Unser Ziel ist es $a = \operatorname{dlog}_g x$ zu berechnen. Wir betrachten eine zufällige Folge von Elementen $b_0, b_1, b_2, \ldots \in_R \{0, 1, 2\}$ und wählen $u_0, v_0 \in_R \mathbb{Z}_d$. Dann setzen wir $y_0 := g^{u_0} \cdot x^{v_0}$ und berechnen y_{k+1} für $k \geq 0$ über die folgende Iterationsvorschrift

$$y_{k+1} := \begin{cases} g \cdot y_k & \text{falls } b_k = 0 \\ y_k^2 & \text{falls } b_k = 1 \\ x \cdot y_k & \text{falls } b_k = 2 \end{cases}$$

bis wir eine Kollision $y_i = y_j$ mit $i \neq j$ gefunden haben. Die Exponenten von x und g in y_k können wir in Form von Polynomen speichern. Dazu setzen wir $\tau_0 := u_0 + v_0 \cdot t$ und

$$\tau_{k+1} := \begin{cases} \tau_k + 1 & \text{falls } b_k = 0 \\ 2 \cdot \tau_k & \text{falls } b_k = 1 \\ \tau_k + t & \text{falls } b_k = 2 \end{cases}$$

Dann gilt $\tau_k = v_k \cdot t + u_k$ und $y_k = g^{\tau_k(a)}$ für alle k, denn $y_k = g^{u_k} \cdot x^{v_k} = g^{u_k} \cdot (g^a)^{v_k} = g^{u_k + a \cdot v_k} = g^{\tau_k(a)}$. Im Fall der Kollision $y_i = y_j$ für $i \neq j$ erhalten wir

$$g^{u_i + a \cdot v_i} = g^{u_i} \cdot x^{v_i} = y_i = y_j = g^{u_j} \cdot x^{v_j} = g^{u_j + a \cdot v_j} \Longrightarrow u_i + a \cdot v_i \equiv u_j + a \cdot v_j \bmod d$$

Gilt also $\mathrm{ggT}(v_i - v_j, d) = 1$, so können wir $v_i - v_j$ modulo d invertieren und erhalten

$$a = (u_j - u_i) \cdot (v_i - v_j)^{-1} \bmod d.$$

Wir müssen uns nun die Frage stellen, wie wahrscheinlich der Fall ist, dass wir $v_i - v_j$ in der obigen Situation modulo d tatsächlich invertieren können, und wie wir verfahren, wenn $v_i - v_j$ eben nicht invertierbar ist.

Bemerkung 3.22. Wir möchten eine Gleichung der Form

$$m_1 - m_2 \equiv k \cdot (b_1 - b_2) \bmod d$$

nach k auflösen. Falls $1 = e := \mathrm{ggT}(b_1 - b_2, d)$ gilt, so können wir wie oben bereits schlicht mit dem Inversen von $b_1 - b_2$ multiplizieren. Gilt jedoch $e > 1$, so verfahren wir wie folgt: Weil e ein Teiler von $b_1 - b_2$ ist, ist auch $m_1 - m_2$ durch e teilbar und wir können ersatzweise die Gleichung

$$\frac{m_1 - m_2}{e} \equiv k \cdot \frac{b_1 - b_2}{e} \bmod \frac{d}{e}$$

betrachten. Wegen $e = \mathrm{ggT}(b_1 - b_2, d)$ sind $\frac{b_1 - b_2}{e}$ und $\frac{d}{e}$ teilerfremd, so dass wir $\frac{b_1 - b_2}{e}$ modulo $\frac{d}{e}$ invertieren können. Dies liefert uns k' mit

$$k' \equiv \frac{m_1 - m_2}{e} \cdot \left(\frac{b_1 - b_2}{e} \right)^{-1} \bmod \frac{d}{e}.$$

Es gilt also $k \equiv k' \bmod \frac{d}{e}$, d.h. $k - k' = i \cdot \frac{d}{e}$ für ein $i \in \mathbb{Z}$ und folglich

$$k \equiv k' + i \cdot \frac{d}{e} \bmod d$$

für ein i mit $0 \leq i \leq e - 1$. Eine Möglichkeit den richtigen Wert für k herauszufinden ist nun, alle möglichen Werte von i auszuprobieren und zu vergleichen, ob sich das gewünschte Ergebnis ergibt. Für kleine Werte von e ist dieses Verfahren durchaus praktikabel. Insbesondere in dem Spezialfall, dass die Gruppenordnung d eine Primzahl ist, kann e nur die Werte 1 oder d annehmen.

Bemerkung 3.23. Die Wahrscheinlichkeit, dass eine zufällige Zahl aus $\{1, \ldots, d-1\}$ teilerfremd zu d ist, berechnet sich exakterweise zu $\prod_{\substack{p \mid d \\ p \text{ prim}}} (1 - \frac{1}{p})$. Hat d nur sehr große Primfaktoren, so ist die Wahrscheinlichkeit dafür, durch zufällige Wahl aus $\{1, \ldots, d-1\}$ ein zu d teilerfremdes Element zu erhalten, relativ groß. Diese etwas schwammige Aussage soll jedoch nur an die Intuition appellieren. Genauer gilt:

Satz. Die Anzahl der zu d teilerfremden Zahlen ist $\varphi(d)$, wobei φ die *Eulersche φ-Funktion* bezeichnet. Es gilt

$$\varphi(d) \approx \frac{6}{\pi^2} \cdot d \approx 0.6079271019 \cdot d$$

und damit ist die Wahrscheinlichkeit durch zufällige Wahl aus $\{1, \ldots, d-1\}$ eine zu d teilerfremde Zahl zu erhalten näherungsweise 61%.

Eine Wahrscheinlichkeit deutlich größer als 50% soll uns genügen. Die Aussage dieses Satzes entspricht der Aussage über die durchschnittliche Ordnung von $\varphi(n)$ aus dem Buch [10].

Bemerkung 3.24. Zurück zur algorithmischen Idee zur Bestimmung des diskreten Logarithmus, die wir in der ersten Bemerkung dieses Abschnitts skizziert hatten. Auch wenn die Sequenz der $(y_i)_{i\in\mathbb{N}}$ keinesfalls zufällig ist, so zeigt jedoch die Praxis, dass sie sich wie "zufällig verhält". Wir werden sie im folgenden wie eine Sequenz von Zufallszahlen behandeln. Der Satz über das Geburtstagsproblem sagt uns, dass wir nach $\mathcal{O}(\sqrt{d})$ Schritten, $d = \#G$, mit einer Kollision rechnen können. Die bisherige algorithmische Idee verlangt allerdings noch immer das Speichern aller Werte b_i, $i = 0, 1, 2, \ldots$, mit deren Hilfe sich dann die entsprechenden Polynome τ_i berechnen lassen und mit deren Hilfe dann wiederum der diskrete Logarithmus ermittelt werden kann. Als erste Verbesserung werden wir Floyd's Zykel Trick 2.5 in den Algorithmus einfügen. Dann partitionieren wir G in drei ungefähr gleich große Teilmengen S_0, S_1 und S_2 und ändern die Iterationsvorschriften wie folgt ab:

$$y_{k+1} := \begin{cases} g \cdot y_k & \text{falls } y_k \in S_0 \\ y_k^2 & \text{falls } y_k \in S_1 \\ x \cdot y_k & \text{falls } y_k \in S_2 \end{cases} \qquad \tau_{k+1} := \begin{cases} \tau_k + 1 & \text{falls } y_k \in S_0 \\ 2 \cdot \tau_k & \text{falls } y_k \in S_1 \\ \tau_k + t & \text{falls } y_k \in S_2 \end{cases}$$

Nun muss die Sequenz der b_i nicht mehr gespeichert werden, denn wir haben die zufällige Wahl durch eine deterministische ersetzt, die in der Praxis ähnlich gute Resultate liefert, wie eine zufällige Wahl.

Nun kommen wir endlich zu dem eigentlichen Algorithmus:

Algorithmus 3.25. (Die Pollard ρ-Methode für DLP) Sei $G = \langle g \rangle$, $d = \#G$ und $x \in G$. Ferner sei $G = S_0 \uplus S_1 \uplus S_2$ eine Partition von G mit $|S_0| \approx |S_1| \approx |S_2|$.

(1) Wähle $u_0, v_0 \in_R \mathbb{Z}_d$ und setze $x_0 := g^{u_0} \cdot x^{v_0}$, $y_0 := g^{u_0} \cdot x^{v_0}$, $\tau_0 := u_0 + t \cdot v_0$, $\sigma_0 := u_0 + t \cdot v_0$, $k := 0$.

(2) Wiederhole die folgenden Schritte, bis eine Kollision $x_k = y_k$, $k > 0$, auftritt:

(i) Berechne

$$x_{k+1} := \begin{cases} g \cdot x_k & \text{falls } x_k \in S_0 \\ x_k^2 & \text{falls } x_k \in S_1 \\ x \cdot x_k & \text{falls } x_k \in S_2 \end{cases} \qquad \tau_{k+1} := \begin{cases} \tau_k + 1 & \text{falls } x_k \in S_0 \\ 2 \cdot \tau_k & \text{falls } x_k \in S_1 \\ \tau_k + t & \text{falls } x_k \in S_2 \end{cases}$$

(ii) Berechne

$$y_{k+1}^{\mathsf{tmp}} := \begin{cases} g \cdot y_k & \text{falls } y_k \in S_0 \\ y_k^2 & \text{falls } y_k \in S_1 \\ x \cdot y_k & \text{falls } y_k \in S_2 \end{cases} \qquad \sigma_{k+1}^{\mathsf{tmp}} := \begin{cases} \sigma_k + 1 & \text{falls } y_k \in S_0 \\ 2 \cdot \sigma_k & \text{falls } y_k \in S_1 \\ \sigma_k + t & \text{falls } y_k \in S_2 \end{cases}$$

und

$$y_{k+1} := \begin{cases} g \cdot y_{k+1}^{\mathsf{tmp}} & \text{falls } y_{k+1}^{\mathsf{tmp}} \in S_0 \\ (y_{k+1}^{\mathsf{tmp}})^2 & \text{falls } y_{k+1}^{\mathsf{tmp}} \in S_1 \\ x \cdot y_{k+1}^{\mathsf{tmp}} & \text{falls } y_{k+1}^{\mathsf{tmp}} \in S_2 \end{cases} \qquad \sigma_{k+1} := \begin{cases} \sigma_{k+1}^{\mathsf{tmp}} + 1 & \text{falls } y_{k+1}^{\mathsf{tmp}} \in S_0 \\ 2 \cdot \sigma_{k+1}^{\mathsf{tmp}} & \text{falls } y_{k+1}^{\mathsf{tmp}} \in S_1 \\ \sigma_{k+1}^{\mathsf{tmp}} + t & \text{falls } y_{k+1}^{\mathsf{tmp}} \in S_2 \end{cases}$$

(iii) Setze $k := k + 1$.

(3) Gilt $x_k = g^{u_k} \cdot x^{v_k} = g^{s_k} \cdot x^{t_k} = y_k$ und $\mathrm{ggT}(v_k - t_k, d) = 1$, so gebe $(s_k - u_k) \cdot (v_k - t_k)^{-1} \bmod d$ aus. Anderenfalls gebe FAIL zurück.

Die Polynome können während der Berechnung ruhig modulo d reduziert werden, um die Koeffizienten möglichst klein zu halten. Insgesamt erhalten wir also:

Theorem 3.26. *Die Pollard ρ-Methode für den diskreten Logarithmus 3.25 berechnet mit einer erwarteten Anzahl von $\mathcal{O}(\sqrt{d})$ Gruppenoperationen, $d = \#G$, und konstantem Speicherbedarf den diskreten Logarithmus.*

Beweis. Folgt aus dem Satz über die erwartete Anzahl von Wahlen bis zur ersten Kollision unter der Annahme, dass sich die Sequenzen der x_i und y_i wie zufällig verhalten. □

3.7 Das Kryptosystem von El'Gamal

Das von El'Gamal 1985 entwickelte Kryptosystem, das wir in diesem Abschnitt kennenlernen werden, basiert im wesentlichen auf dem Schlüsselaustauschverfahren von Diffie & Hellman. El'Gamal schlug eine Erweiterung des Protokolls vor, die es nicht nur ermöglicht, einen geheimen gemeinsamen Schlüssel auszutauschen, sondern zusätzlich auch Nachrichten zu ver- bzw. entschlüsseln:

Protokoll 3.27. (Das Kryptosystem von El'Gamal) Sei G eine zyklische Gruppe mit erzeugendem Element g, d.h. $G = \langle g \rangle$, sowie $d = \#G$.

(1) Alice wählt ein $s_A \in_R \{2, \ldots, d-1\}$ und sendet $x := g^{s_A}$ an Bob.

(2) Bob wählt ebenfalls ein $s_T \in_R \{2, \ldots, d-1\}$ und berechnet $y := g^{s_T}$ (temporärer Schlüssel für einen Nachrichtenaustausch).

(3) Bob möchte Alice die Nachricht $m \in G$ übersenden. Er berechnet $\tilde{m} := m \cdot x^{s_T}$ und sendet $(\tilde{m}, y)$ an Alice.

(4) Bei Erhalt von $(\tilde{m}, y)$ berechnet Alice y^{s_A} und $m = \tilde{m} \cdot (y^{s_A})^{-1}$.

Beweis. (*Korrektheit*) Die Korrektheit folgt aus der Korrektheit des Schlüsselaustauschverfahrens von Diffie & Hellman. □

Zur Effizienz ist zu sagen, dass das Kryptosystem von El'Gamal im wesentlichen die gleiche Laufzeit wie das Schlüsselaustauschverfahren von Diffie & Hellman hat. Dazu kommen eine weitere Multiplikation für Bob zum Verschlüsseln und eine Multiplikation sowie eine Inversion für Alice. All diese Operation sind in einer entsprechend gewählten Gruppe (z.B. $\mathbb{Z}_p^\times$) hinreichend effizient durchführbar.

3.8 "Secret Sharing"

Angenommen, n Personen $\{1, \ldots, n\}$ möchten ein Geheimnis s untereinander aufteilen derart, dass sie nur alle gemeinsam das Geheimnis enthüllen können, aber keine echte Teilmenge der Gruppe. Eine Möglichkeit, ein Geheimnis auf diese Weise auf n Personen "zu verteilen", ist die folgende: Wir fassen s als Element des Körpers $\mathbb{Z}_p$ für eine große Primzahl p (z.B. 1024-Bit) auf. Dann gehen wir wie folgt vor:

Algorithmus 3.28. (Secret Sharing) Sei p eine Primzahl und $s \in \mathbb{Z}_p^\times$ ein zu teilendes Geheimnis.

(1) Wir berechnen n Zufallswerte $x_1, \ldots, x_n \in_R \mathbb{Z}_p$ mit $x_i \neq 0$ und $x_i \neq x_j$ für alle $i \neq j$ (falls zwei Werte übereinstimmen, verwerfen wir einen der beiden und wählen erneut – dieses Verfahren führt rasch zum Ziel, denn eine Kollision ist erst nach $\mathcal{O}(\sqrt{p})$ Zufallswahlen zu erwarten).

(2) Wir berechnen weitere $n-1$ Zufallswerte $a_1, \ldots, a_{n-1} \in_R \mathbb{Z}_p$ mit $a_i \neq 0$.

(3) Wir berechnen für $1 \leq i \leq n$ die Funktionswerte $y_i := p(x_i)$ durch Einsetzen der Werte x_i in das Polynom $p(x) := a_{n-1}x^{n-1} + \ldots + a_1 x + s$ (s ist das zu "teilende" Geheimnis).

(4) Wir übergeben der i-ten Person das "Teilgeheimnis" (x_i, y_i) für $1 \leq i \leq n$.

Mit Hilfe der üblichen *Langrangeschen-Interpolationsformel* können in der Tat alle n Personen zusammen das Polynom $p(x)$ rekonstruieren:

$$p(x) = \sum_{i=1}^{n} y_i \cdot \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Finden sich nun z.B. die Personen $1, \ldots, n-1$ zusammen, um gemeinsam ohne die n-te Person das Geheimnis zu lüften, so können sie dasjenige eindeutig bestimmte Interpolationspolynom $q(x)$ mit $q(x_i) = y_i$ für $i = 1, \ldots, n-1$ bestimmen. Ferner erhalten sie damit eine Darstellung für $p(x)$ der Form $p(x) = q(x) + r(x) \cdot \prod_{i=1}^{n-1}(x - x_i)$. Wegen $\deg p(x) = n-1$ und $\deg q(x) = n-2$ sowie $\deg \prod_{i=1}^{n-1}(x - x_i) = n-1$ können sie $\deg r(x) = 0$, also $r(x) = c \in \mathbb{Z}_p$

folgern. Da aber jeder Wert in $\mathbb{Z}_p$ gewählt werden kann, ohne dass die Bedingungen $p(x_i) = y_i$ für $i = 1, \ldots, n-1$ verletzt werden, können sie keinerlei Rückschlüsse auf das tatsächliche Geheimnis s ziehen. Trifft sich dagegen eine Koalition von noch weniger als $n-1$ Teilnehmern, so wächst mit dem Grad des Polynoms $r(x)$ in $p(x) = q(x) + r(x) \cdot \prod_i (x - x_i)$ auch die Anzahl der Freiheitsgrade für seine Koeffizienten. Die Chance, das Geheimnis zu enthüllen, wird noch weitaus geringer.
Das beschriebene Verfahren kann so modifiziert werden, dass ein Geheimnis auf n Personen verteilt wird, von denen dann eine Teilgruppe von genau $k \leq n$ Personen in der Lage ist, es zu enthüllen: Statt das Polynom $p(x)$ vom Grad $n-1$ zu wählen, berechnet man ein Polynom vom Grad $k-1$ und verteilt die entsprechenden Argumente und Funktionswerte wie oben beschrieben. Ein Polynom $(k-1)$-ten Grades kann dann von k Personen mit Hilfe der Argumente und der Funktionswerte über die Lagrangesche Interpolationsformel bestimmt werden.

Übungsaufgaben

Einfache Aufgaben

Übungsaufgabe 1. Berechnen Sie den größten gemeinsamen Teiler und das kleinste gemeinsame Vielfache der Zahlen $a = 123425$ und $b = 364625$. Finden Sie die Zahlen k, l, die der Euklidische Algorithmus liefert, so dass gilt: $\mathsf{ggT}(a, b) = k \cdot a + l \cdot b$.

MuPAD-Befehle:

- ✓ `igcd`
- ✓ `igcdex`
- ✓ `ilcm`

Übungsaufgabe 2. Wir betrachten jetzt statt ganzer Zahlen a, b Polynome

$$p = 4x^4 - 44x^3 + 164x^2 - 244x + 120 \quad \text{und} \quad q = -2x^3 + 10x^2 - 4x - 16.$$

Berechnen Sie wie in der vorhergehenden Aufgabe den größten gemeinsamen Teiler, Polynome r, s, so dass

$$\mathsf{ggT}(p, q) = r \cdot p + s \cdot q,$$

und das kleinste gemeinsame Vielfache der beiden Polynome. Das Polynom $2x + 1$ z.B. gibt man in MuPAD mittels `poly(2*x + 1, [x])` ein.

MuPAD-Befehle:

- ✓ `poly`
- ✓ `gcd`
- ✓ `gcdex`
- ✓ `lcm`

Übungsaufgabe 3. MuPAD bietet mit der Funktion `mod` die Möglichkeit zum modularen Rechnen. So ist die Zahl `a mod b` der nichtnegative, ganzzahlige Rest bei Division von `a` durch `b`. Berechnen Sie 23524 mod 3545 und 226264 mod 36366.

MuPAD-Befehle:

- ✓ `mod`

Übungsaufgabe 4. Bei kryptographischen Verfahren wie z.B. dem RSA-Verfahren ist schnelles Potenzieren modulo einer sehr großen Zahl ungeheuer wichtig. Wir betrachten die Zahl

$$N = 23942934792853 \cdot 3483490509234257.$$

Mit `powermod(35463, 2535647, N)` berechnet man $35463^{2535647}$ mod N. Berechnen Sie das Ergebnis mit MuPAD – keine Angst, die Rechnung wird nicht viel Zeit in Anspruch nehmen (woran könnte dies liegen?).

MuPAD-Befehle:

- ✓ `powermod`

Übungsaufgabe 5. Auf den französischen Mathematiker *Pierre de Fermat* gehen die *Fermatschen Primzahlen* zurück. Dies sind genau die Primzahlen p, die sich in der Form $p = 2^{2^n} + 1$, $n \in \mathbb{N}_0$, schreiben lassen. Fermat wußte bereits, dass für $n = 0, 1, 2, 3, 4$ die Zahlen $2^{2^n} + 1$ prim sind. Er vermutete, dass sie für alle n prim sind. Diese Vermutung erwies sich allerdings bereits für $n = 5$ als falsch.

1. Zeigen Sie mit Hilfe der Funktion `isprime`, dass $2^{2^n} + 1$ für $n = 0, 1, 2, 3, 4$ tatsächlich eine Primzahl ist, dass dies aber für $n = 5$ nicht der Fall ist. Bei der Größe der Zahl für $n = 5$ sollte klar sein, warum Fermat zu seiner Zeit nicht einfach alle Teiler der Zahl durchprobieren konnte, um festzustellen, dass sie nicht prim ist.

2. Berechnen Sie die Primfaktorzerlegung der Zahl $2^{2^5} + 1$ mit Hilfe der Funktion `factor`. Wie viele Zahlen hätte Pierre de Fermat durchprobieren müssen, um den ersten echten Teiler von $2^{2^5} + 1$ zu finden?

3. Bestimmen Sie die nächste Primzahl nach $2^{2^4} + 1$.

4. Bestimmen Sie zwei Primzahlen, so dass das Produkt der Primzahlen eine Zahl ist, die etwa in der Größenordnung von $5.000.000.000$ liegt.

5. Bestimmen Sie mit MuPAD die $999.$, $9999.$ und die $99999.$ Primzahl.

MuPAD-Befehle:

- ✓ `isprime`
- ✓ `factor`

- ✓ `nextprime`
- ✓ `ithprime`

Übungsaufgabe 6. MuPAD bietet mit dem Domain `Dom::IntegerMod(n)` einen Koeffizientenbereich, der das Rechnen in dem Restklassenring modulo der Zahl `n` ermöglicht. Definieren Sie den Ringe

`R1:= Dom::IntegerMod(12)` und `R2:= Dom::IntegerMod(13)`

Mit dem Befehl `R1(2)` können Sie sich nun das Element 2 mod 12 in MuPAD erzeugen.

1. Erzeugen Sie die Elemente 5 mod 12, 6 mod 13 und 14 mod 12 (letzteres ist kein Durckfehler).
2. Berechen Sie für $a := 5$ mod i und $b := 7$ mod i die Summe und das Produkt von a und b für $i = 12, 13$.
3. Beweisen Sie experimentell, dass `R2` ein Körper ist, `R1` dagegen nicht. (Welche Elemente sind wo invertierbar?)
4. Wir können jetzt auch Polynome mit Koeffienten in `R1` oder `R2` definieren. Die Zeile

   `p:= poly(2*x + 1, [x], R1)`

   liefert uns in MuPAD das Polynom $p = (2 \text{ mod } 12)x + (1 \text{ mod } 12)$. Definieren Sie die Polynome

   $$p := (4 \text{ mod } 12)x^3 + (3 \text{ mod } 12)x \quad \text{und} \quad q := (5 \text{ mod } 12)x^4 + (6 \text{ mod } 12)$$

   in MuPAD und berechnen Sie $p + q$ und $p \cdot q$ sowie p^4.

MuPAD-Befehle:

- ✓ `Dom::IntegerMod()`
- ✓ `poly`

Weiterführende Aufgaben

Im Rahmen der folgenden Aufgaben sollen die wichtigsten der bisher vorgestellten Algorithmen in MuPAD implementiert werden.

Übungsaufgabe 7. Implementiere den Algorithmus 1.5 zum wiederholten Quadrieren in MuPAD. Finde dazu zuerst die MuPAD-Funktion, die die Binärdarstellung einer natürlichen Zahl liefert. Suche diese Funktion in der `numlib`-Bibliothek. Achte besonders darauf, wie die Rückgabe der Funktion angeordnet ist. Prüfe, ob Dein Algorithmus auch über Restklasseningen in MuPAD korrekt funktioniert. Wirf, wenn nötig, einen Blick auf die Hilfeseite von `Dom::IntegerMod`. Prüfe Deine Implementation durch Vergleich mit der Systemfunktion `powermod`.

Übungsaufgabe 8. Implementiere Algorithmus den Erweiterten Euklidischen Algorithmus 1.14 in MuPAD. Lies dazu, falls nötig, die Hilfeseiten zum `while`-Konstrukt in MuPAD. Prüfe anschließend Deine Implementation durch Vergleich mit der Systemfunktion `igcd` bzw. igcdex.

Übungsaufgabe 9. Implementiere Algorithmus 1.21 zur Berechnung des modularen Inversen in MuPAD. Schreibe den Algorithmus so, dass er auf nicht-negativen ganzen Zahlen arbeitet. Verwende Deine Implmentation des Erweiterten Euklidischen Algorithmus. Prüfe Deine Implementation mit Hilfe von `Dom::IntegerMod` an einigen Beispielen.

Übungsaufgabe 10. Implementiere den Fermat-Test 2.2 in MuPAD. Informiere Dich über die Funktion `random` in MuPAD. Benutze diese Funktion zur Wahl der Zufallselemente in Schritt 2 (i) von Algorithmus 2.2. Verwende Deine Implementation des Erweiterten Euklidischen Algorithmus zur Berechnung des größten gemeinsamen Teilers in Schritt 2 (ii) und Deine Implementation des Repeated Squarings in Schritt 2 (iii) von Algorithmus 2.2. Prüfe Deine Implementation an einige Beispielen. Die MuPAD-Systemfunktion `isprime` ist vielleicht hilfreich.

Übungsaufgabe 11. Implementiere das Sieb des Erathostenes 2.3 in MuPAD. Berechne die ersten 1000 Primzahlen. Prüfe Deine Implementation mit Hilfe einer entsprechenden MuPAD-Systemfunktion aus der Bibliothek `numlib` zur Zahlentheorie.

Übungsaufgabe 12. Implementiere den Faktorisierungsalgorithmus Pollard's ρ-Methode 2.9 in MuPAD. Prüfe Deine Implementation anhand einiger kleinerer Beispiele und durch Vergleich mit der Systemfunktion `ifactor` in MuPAD. Gibt es in MuPAD eine Implementation der Pollard ρ-Methode? Vesuche, die Zahl $100000000190000000039000000741$ mit Deiner Implementation der Pollard ρ-Methode zu faktorisieren.

Übungsaufgabe 13. Implementiere das RSA-Verfahren 3.1 in MuPAD. Modularisiere das Verfahren dabei wie folgt: Schreibe eine Routine zur Berechnung des öffentlichen Schlüssels (N, e) und des privaten Schlüssels. Schreibe dann je eine Routine, die das Verschlüsseln bzw. das Entschlüsseln einer Nachricht erledigen.

Übungsaufgabe 14. Lade das MuPAD-Notebook `RSA-Mathe-am-Computer` von der Webseite der Veranstaltung herunter und bearbeite die dort vorgesehenen Aufgabenstellungen.

Übungsaufgabe 15. Implementiere den naiven Angriff 3.2 auf RSA in MuPAD. Verwende als Primzahltest einmal Deine Implementation des Fermat-Tests 2.2 und einmal die Systemfunktion `isprime`. Prüfe Deine Implementation, indem Du mit Hilfe von `nextprime` zwei Primzahlen p und q berechnest, die unmittelbar aufeinander folgen. Wähle den zu faktorisierenden RSA-Modul N als Produkt von p und q.

Übungsaufgabe 16. Implementiere den Algorithmus 3.6 zur Berechnung der Kettenbruchzerlegung in MuPAD. Gibt es eine Systemfunktion mit entsprechender Funktionalität in MuPAD? Prüfe Deine Implementation an Beispielen.

Übungsaufgabe 17. Implementiere den Angriff von Wiener 3.13 auf RSA in MuPAD. Verwende zur Berechnung der Konvergenten der Kettenbruchentwicklung Deine Implementation von Algorithmus 3.6. Prüfe Deine Implementation an dem Beispiel

$$N = 87209474417398506218473228057873724205564460314966454593704581.$$

Wie lange braucht Dein Algorithmus zur Faktorisierung von N im Vergleich zur Systemfunktion `ifactor`?

Übungsaufgabe 18. Implementiere das Schlüsselaustauschverfahren von Diffie & Hellman 3.14 für $G = \mathbb{Z}_p^\times$, p prim, in MuPAD. Teste Deine Implementation an einem Beispiel. Bedenke, dass Du ein erzeugendes Element g der Gruppe $\mathbb{Z}_p^\times$ benötigst. Verwende $g = 32235 \mod 10000000019$. Weise mit einer entsprechenden Funktion in Bibliothek `numlib` nach, dass die Ordnung von g in der Tat 10000000018 ist.

Übungsaufgabe 19. Implementiere den Baby-Schritt-Riesen-Schritt Algorithmus 3.16 zur Berechnung des diskreten Logarithmus in einer endlichen zyklischen Gruppe. Teste Deine Implementation an einigen Gruppenelementen aus der Gruppe $G = \mathbb{Z}_p^\times$ mit $p = 9803$ und dem erzeugenden Element $g = 332 \mod 9803$. Bei Interesse gibt der Aufruf `bytes()` in MuPAD Auskunft über den Speicherbedarf.

Übungsaufgabe 20. Implementiere den Geburtstagsangriff auf den diskreten Logarithmus 3.19. Teste Deine Implementation an einigen Gruppenelementen aus der Gruppe $G = \mathbb{Z}_p^\times$ mit $p = 9803$ und dem erzeugenden Element $g = 332 \mod 9803$.

Übungsaufgabe 21. Implementiere die Pollard-ρ-Methode 3.25 zur Berechnung des diskreten Logarithmus in einer endlichen zyklischen Gruppe. Teste Deine Implementation an einigen Gruppenelementen aus der Gruppe $G = \mathbb{Z}_p^\times$ mit $p = 9803$ und dem erzeugenden Element $g = 332 \mod 9803$. Verbessere anschließend Deine Implementation durch Verwendung der Ideen aus Bemerkung 3.22 für den Fall, dass $v_k - t_k$ modulo d nicht invertierbar ist.

Übungsaufgabe 22. Implementiere das Kryptosystem von El'Gamal 3.27 für $G = \mathbb{Z}_p^\times$, p prim, in MuPAD. Schreibe eine Routine zur Erzeugung des öffentlichen Schlüssels x, eine Routine zum Verschlüsseln einer Nachricht und eine weitere Routine zum Entschlüsseln einer Nachricht. Verwende Deine Implementation des Repeated Squaring 1.5 in den Schritten (1), (2), (3) und (4) sowie Deine Implementation des Algrotihmus zur Berechnung des modularen Inversen 1.21 in Schritt (4). Teste Deine Implementation für $g = 32235 \mod 10000000019$ (gleiche Werte wie bei der Implementation des Schlüsselaustauschverfahrens von Diffie & Hellman).

Übungsaufgabe 23. Implementiere den Secret-Sharing-Algorithmus 3.28 in MuPAD. Schreibe dazu einer Routine, die ein Geheimnis aufteilt und eine weitere Routine, die mit Hilfe der Lagrange'schen Interpolationsformel das Geheimnis aus den Teilgeheimnissen rekonstruiert.

Übungsaufgabe 24. Implementiere den Algorithmus **Dixon's Zufallsquadrate** in MuPAD.

Algorithmus (Dixon's Zufallsquadrate) Seien $N, B > 0$ gegeben.

(1) Berechne (z.B. mit dem Sieb des Erathostenes 2.3) alle Primzahlen $p_1, \ldots, p_h$ aus $\{0, \ldots, B\}$.

(2) Setze $A := \varnothing$ und wiederhole die folgenden Schritte bis $\#A = h + 1$.

(i) Wähle $b \in_R \{0, \ldots, N-2\}$. Falls $g := \mathrm{ggT}(b, N) > 1$, so gebe g aus.

(ii) Setze $a := b^2 \bmod N$ und faktorisiere a über der Faktorbasis $\{p_1, \ldots, p_h\}$. Falls a nicht faktorisiert, verwerfe a und gehe zurück zu (i).

(iii) Ist $a = p_1^{\alpha_1} \cdot \ldots \cdot p_h^{\alpha_h}$, so setze $\alpha := (\alpha_1, \ldots, \alpha_h)$ und $A := A \cup \{(b, \alpha)\}$.

(3) Sei $A = \{(b_1, \alpha^{(1)}), \ldots, (b_{h+1}, \alpha^{(h+1)})\}$. Dann gilt:

$$\begin{aligned} b_1^2 &\equiv p_1^{\alpha_1^{(1)}} \cdot \ldots \cdot p_h^{\alpha_h^{(1)}} \bmod N \\ &\vdots \\ b_{h+1}^2 &\equiv p_1^{\alpha_1^{(h+1)}} \cdot \ldots \cdot p_h^{\alpha_h^{(h+1)}} \bmod N \end{aligned}$$

Löse nun das lineare Gleichungssystem

$$\begin{pmatrix} \alpha_1^{(1)} & \alpha_1^{(2)} & \ldots & \alpha_1^{(h+1)} \\ \vdots & \vdots & & \vdots \\ \alpha_h^{(1)} & \alpha_h^{(2)} & \ldots & \alpha_h^{(h+1)} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_{h+1} \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

über $\mathbb{Z}_2$. Sei $(x_1, \ldots, x_{h+1})$ eine nichttriviale Lösung des obigen Gleichungssystems (beachte, dass es eine solche nach Wahl der Koeffizientenmatrix geben muss, denn diese hat höchstens Rang h). Sind dann $x_{i_1}, \ldots, x_{i_k}$ die von Null verschiedenen Komponenten des Vektors $(x_1, \ldots, x_{h+1})$, so gilt $\alpha^{(i_1)} + \ldots + \alpha^{(i_k)} \equiv 0 \bmod 2$. Setze $x := b_{i_1} \cdot \ldots \cdot b_{i_k}$ und $y := p_1^{\gamma_1} \cdot \ldots \cdot p_h^{\gamma_h}$, wobei $\gamma = \frac{1}{2} \cdot (\alpha^{(i_1)} + \ldots + \alpha^{(i_k)})$.

(4) Wegen $b_{i_j}^2 \equiv p_1^{\alpha_1^{(i_j)}} \cdot \ldots \cdot p_h^{\alpha_h^{(i_j)}} \bmod N$ gilt nun $x^2 \equiv y^2 \bmod N$. Gebe $\mathrm{ggT}(x+y, N)$ und $\mathrm{ggT}(x-y, N)$ aus.

Literatur

[1] J. Blömer, *Vorlesungen zum RSA-Verfahren*, Universität Paderborn, 2002

[2] J. Blömer, *Vorlesungen zu Algorithmen in der Zahlentheorie*, Universität Paderborn, 2002

[3] S. Bosch, *Algebra*, Springer Lehrbuch, 3. Auflage, Springer-Verlag, 1999

[4] J. A. Buchmann, *Introduction to Cryptography*, Undergraduate Texts in Mathematics, second edition, Springer-Verlag, 2001

[5] P. Bundschuh, *Zahlentheorie*, Springer Lehrbuch, 2. Auflage, Springer-Verlag, 1992

[6] H. Cohen, *A Course in Computational Algebraic Number Theory*, Graduate Texts in Mathematics 138, Springer-Verlag, 1993

[7] J. Daemen, V. Rijmen, *AES Proposal: The Rijndael Block Cipher*, Document version 2 vom 03.09.1999, erhältlich im Web

[8] J. von zur Gathen, *Cryptography I*, Skript zu den Vorlesungen zur Kryptographie, Universität Paderborn, Version vom März 2002

[9] J. von zur Gathen & J. Gerhard, *Modern Computer Algebra*, Cambridge University Press 1999

[10] G. Hardy, E. Wright, *Zahlentheorie*, R. Oldenbourg, München, 1958, Übersetzung ins Deutsche des Orginaltitels *An Introduction to the Theory of Numbers* erschienen bei Cambridge University Press

[11] K. Ireland, M. Rosen, *A Classical Introduction to Modern Number Theorie*, Graduate Texts in Mathematics 84, Second Edition, Springer-Verlag, 1990

[12] K.-H. Kiyek, F. Schwarz, *Lineare Algebra*, Teubner Studienbücher Mathematik, B. G. Teubner, 1999

[13] N. Koblitz, *A Course in Number Theory and Cryptograpy*, Graduate Texts in Mathematics 114, Second Edition, Springer-Verlag, 1994

[14] U. Krengel, *Einführung in die Wahrscheinlichkeitstheorie und Statistik*, Vieweg Studium – Aufbaukurs Mathematik, 5. Auflage, Vieweg Verlag, 2000

[15] H. Kurzweil, B. Stellmacher, *Theorie der endlichen Gruppen – Eine Einführung*, Springer Lehrbuch, 1. Auflage, Springer-Verlag, 1998

[16] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, Florida, 1997

[17] G. Scheja, U. Storch, *Lehrbuch der Algebra (Unter Einschluß der Linearen Algebra) Teil 1*, Mathematische Leitfäden, B. G. Teubner, 1994

[18] G. Scheja, U. Storch, *Lehrbuch der Algebra Teil 2*, Mathematische Leitfäden, B. G. Teubner, 1994

[19] A. Weil, *Basic Number Theory*, Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen Vol. 144, Springer-Verlag, 1973